AFRL-RI-RS-TR-2015-250

INDUSTRY STRENGTH TOOL AND TECHNOLOGY FOR AUTOMATED SYNTHESIS OF SAFETY-CRITICAL APPLICATIONS FROM FORMAL SPECIFICATIONS

VIRGINIA POLYTECHNIC INSTITUTE & STATE UNIVERSITY (VIRGINIA TECH)

NOVEMBER 2015

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

FOR THE DIRECTOR:

/ S /	**/ S /**
WILMAR W. SIFRE	MARK H. LINDERMAN
Work Unit Manager	Technical Advisor, Computing
	& Communications Division
	Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* NOV 2015	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* FEB 2013 – JUN 2015

4. TITLE AND SUBTITLE INDUSTRY STRENGTH TOOL AND TECHNOLOGY FOR AUTOMATED SYNTHESIS OF SAFETY-CRITICAL APPLICATIONS FROM FORMAL SPECIFICATIONS	**5a. CONTRACT NUMBER** FA8750-13-C-0053
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 63781D
6. AUTHOR(S) Sandeep K. Shukla, Mahesh Nanjundappa, Matthew Anderson, Avik Dayal and Mattthew Kracht	**5d. PROJECT NUMBER** ASET
	5e. TASK NUMBER 12
	5f. WORK UNIT NUMBER VT
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Virginia Polytechnic Institute & State University (Virginia Tech) 1880 Pratt Drive STE 2006 Blacksburg VA 24060-6750	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA 525 Brooks Road Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2015-250

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2015-5454
Date Cleared: 9 Nov 15

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report discussed the fundamental theory, algorithms, and prototype tools for the synthesis of embedded safety-critical software for multi-core embedded platforms as well as to initiate planned technology transfer to a Department of Defense (DoD) vender company. The techniques and tools developed during project execution will enable the production of embedded safety-critical software with improved quality and performance, decreases in cost and development times, automation of labor-intensive and error-prone processes, increased reliability, and easier integration and sustainment.

15. SUBJECT TERMS

Software Engineering, Software Producibility, Component-based software design, behavioral types, behavioral type interference, Polychronous model of computation, Prime Implicates, Boolean Abstraction, real-time embedded software, software synthesis, correct by construction software design, model-driven software design, high-assurance software.

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**WILMAR W. SIFRE**
U	U	U	UU	155	**19b. TELEPHONE NUMBER** (*Include area code*) **N/A**

Table of Contents

List of Figures v

List of Tables vii

Summary 1

1 Introduction 4

1.1Software Synthesis 4

1.2 Hardware Synthesis 7

1.3 Verification and Validation 8

1.3.1 Causality Analysis 9

1.3.2 Type Consistency Analysis 9

1.3.3 Value Range Analysis 10

1.4 Summary of contributions 10

1.5 Organization of the Report 12

1.6 Publications on the work reported in this report 14

2 Methods, Assumptions and Procedures 15

2.1 Synchronous and Polychronous Model of Computation 16

2.2Synchronous Languages 18

2.2.1 Esterel 18

2.2.2 Quartz 20

2.2.3 Lustre 20

2.2.4 Signal 22

2.2.5 Statecharts, SyncCharts, Argos, Reactive-C 23

2.3 Alternatives to Synchronous languages 25

2.3.1 Kahn Process Networks (KPN) 26

2.3.2 Synchronous Dataflow (SDF) and Multi-dimensional (MD-SDF) 27

2.3.3 Petri Nets 28

2.3.4 Input/Output Automata 29

2.3.5 Ptolemy Framework 29

3 Results and Discussion 31

3.1 Related Work 31

3.1.1 Concurrent Software Synthesis 31

3.1.2 ASIP Synthesis 32

3.1.3 Verification and Validation 32
3.1.3.1 Causality Analysis 32
3.1.3.2 Type Checking 33
3.1.3.3 Value Range Analysis 34
3.2 MRICDF Polychronous Formalism and EmCodeSyn Synthesis Tool 35
3.2.1 Definitions and Overview of Concepts 35
3.2.2 MRICDF Actors 37
3.2.3 Master Trigger and Sequential Implementability 38
3.3 Synthesis of Multi-Threaded Code from Polychronous models 38
3.3.1 Constraints for Concurrent Implementability 44
3.3.2 Computing Partial Triggers 45
3.3.3 Constructing the forest of clock trees T 47
3.3.4 Check for Data Dependencies and Deadlock 48
3.3.5 Identification of Shared Epochs 49
3.3.6 Mapping and Multi-threaded Code Generation 49
3.3.7 Experimental Evaluation and Discussions 50
3.3.8 Summary 53
3.4 Synthesis of Improved Multi-threaded Code from Polychronous models based on Analysis of Affine Relations 54
3.4.1 Avoidable Synchronizations 54
3.4.2 Affine Transformations and Affine Relations in Polychronous Languages 57
3.4.2.1 Affine Transformations and Relations 57
3.4.2.2 Constructs to express Affine Transformations and Relations 58
3.4.3 Analysis of Affine Relations for Improved Multi-threaded Code Synthesis 59
3.4.4 Summary 63
3.5 Synthesis of Application-Specific Instruction-set Processor(ASIP) from Polychronous models 64
3.5.1 Conditional Partial Order Graphs 64
3.5.1.1 Definition: Conditional Partial Order Graph 64
3.5.2 MRICDF Actors and their CPOGs 66
3.5.2.1 Example MRICDF model and it’s CPOG 73
3.5.3 Transformations, Resource Estimation and Implementability 76
3.5.3.1 Resource Estimation 78

3.5.3.2 Implementability 79
3.5.4 Summary 80
3.6 Polyhedral Model based Causality Detection in Polychronous models 81
3.6.1 Analysis of Polychronous Specifications 83
3.6.2 SMT extension to Polychrony 85
3.6.3 Polyhedral Model based Analysis 86
3.6.3.1 Constraint Extraction and Transformation 87
3.6.3.2 Polyhedral Analysis 87
3.6.3.3 Limitation of Polyhedral libraries 88
3.6.3.4 Safe code synthesis using Wrapper 88
3.6.3.5 Implementation and Design Flow 89
3.6.4 Summary 90
3.7 Type Inference and Type Consistency Verification of Polychronous models 91
3.7.1 Type System and Inference Rules 93
3.7.1.1 Type Set 94
3.7.1.2 Inference Algorithm 97
3.7.2 SMT based Type Consistency Checking 98
3.7.3 Summary 103
3.8 SMT based Value Range Analysis of Polychronous models 103
3.8.1 Range Inference Rules and Analysis 106
3.8.1.1 Definitions 107
3.8.1.2 Widening operation (∐) 107
3.8.1.3 Narrowing Operation 109
3.8.1.4 Value Range Inference Rules 109
3.8.2 SMT based Verification of properties related to Signal Value Ranges 112
3.8.2.1 SMT constraints for Constants 112
3.8.2.2 SMT constraints for Sampler actors 112
3.8.3 SMT constraints for Merge actors 113
3.8.4 SMT constraints for Function actors 113
3.8.5 SMT constraints for Buffer actors 113
3.8.6 Case Study: Automated Bathtub System 114
3.8.3 Summary 118
4 Conclusions and Future Work 120

4.1 Conclusions 120
4.1.1 Software Synthesis techniques 120
4.1.2 Hardware Synthesis techniques 120
4.1.3 Verification and Validation techniques 121
4.2 Future Work 121
4.2.1 Software Synthesis techniques 121
4.2.2 Hardware Synthesis techniques 122
4.2.3 Verification and Validation techniques 122
4.2.4 EmCodeSyn Tool development 122
5 References 124
Appendix 131
Example: Generated Multi-threaded Code 131
List of Symbols, Abbreviations and Acronyms 144

List of Figures

Figure 1: Deployed Volume of Software in Selected Embedded Systems Annually [23] 5
Figure 2: Amount of Software Used in Safety Critical Systems Over Years [23] 5
Figure 3: Flow of ASIP Design Methodology [38] 7
Figure 4: Report Organization 13
Figure 5: Block Diagram of a Mono-clocked System 17
Figure 6: Block Diagram of a Multi-clocked System 17
Figure 7: Statechart representing abstract execution model of an answering machine 24
Figure 8: Execution flow of the Reactive-C program shown in Listing 2.5 25
Figure 9: Example of KPN model 26
Figure 10: Example of Synchronous Dataflow Model 28
Figure 11: Example of Petri net representing water composition 29
Figure 12: (a) MRICDF model, (b) simplified clock tree 44
Figure 13: Pyramid structure of clock tree and forest of clock trees for sequential and concurrent specifications 47
Figure 14: Plot of Time taken for analysis and code generation vs number of times model is duplicated 53
Figure 15: Simple application to illustrate avoidable synchronizations 54
Figure 16: Pseudo-code for the application in Figure 15 55
Figure 17: Sample execution trace of the application in Figure 15 56
Figure 18: Optimized pseudo-code for the application in Figure 15 56
Figure 19: Clock K is (5; 4; 9) affine transformation of clock H 57
Figure 20: Hierarchical Clock Graph 59
Figure 21: Simple Process with 3 sub-processes 60
Figure 22: Example execution trace of the application in Figure 21 when $n = d = 3$, $\phi = 0$ 60
Figure 23: Example execution trace of the application in Figure 21 when $n = d = 3$, $\phi = 2$ 61
Figure 24: Example execution trace of the application in Figure 21 when $n = d = 3$, $\phi = 5$ 61
Figure 25: Example execution trace of the application in Figure 21 when $n = 3, d = 7$, $\phi = 0$ 62
Figure 26: Example execution trace of the application in Figure 21 when $n = 3, d = 7$, $\phi = 2$ 63
Figure 27: Graphical representation of CPOG 66
Figure 28: CPOG for Function Actor 68
Figure 29: CPOG for Buffer Actor 69
Figure 30: CPOG for Sampler Actor 70
Figure 31: CPOG for Merge Actor 71
Figure 32: Sample MRICDF model 74
Figure 33: SIGNAL code (ADD, Comparator, GAIN & $\frac{1}{GAIN}$ are predefined function actors) 74
Figure 34: CPOG for the MRICDF Model 75
Figure 35: Modified CPOG with Boolean vector 1101 and 1110 respectively ($b_{x1} = m, b_{x7} = n$) 78
Figure 36: (Top) 3D-plot (multiple views) of Polyhedrons representing Input and Loop Constraints. $(Bottom)$ 3D plots of $I \cap L$ and $I - L$ 88

Figure 37: Execution flow of enhanced polychrony compiler 90
Figure 38: Example MRICDF model 92
Figure 39: Inference Algorithm 97
Figure 40: Flowchart for Type Inference and SMT Analysis 102
Figure 41: Example MRICDF model and SIGNAL program for Simple Automated Bathtub Embedded System 105
Figure 42: Lattice Structure 108

List of Tables

Table 1: Popular Modeling Languages used for designing Embedded Systems.......................... 15
Table 2: Basic Esterel Statements [12] 18
Table 3: Basic Lustre Operators 21
Table 4: Primitive Signal Operators 22
Table 5: Popular formalisms alternative to Synchronous languages 26
Table 6: Popular Commercial Tools used for Modeling Embedded Systems and Code Generation 30
Table 7: Benchmark Suite 51
Table 8: Experimental Results.. 52
Table 9: Formal representation of CPOG for model in Figure 32.. 75
Table 10: Input and True Causal Loop constraints 87

Summary

A large number of DOD (Department of Defense) software is safety-critical and reactive. Reactive software continually reacts to environmental inputs, and it should always be *responsive* to inputs. Thus reactive software must be able to decide when to expect new inputs, and must react or compute responses at a faster time scale than the time scale of successive inputs. For determinism and compositionality, it is required that the reading of inputs is blocking. Thus, if the software is unable to determine when to read which inputs, it may get blocked waiting to read one input while other inputs may come and get dropped. For responsiveness, one could either design the software to sample all inputs synchronously at the occurrence of a set of regular events (such as clock ticks in case of hardware), or it should have enough internal information to determine when to wait for which inputs. Of course, one way to simplify this problem is to use concurrent threads that read inputs from different input streams. However, concurrency comes at a price – synchronizing these threads for correct merging of data flows or sharing data among threads becomes challenging. *Manually* writing such concurrent reactive programs is *time consuming* and *error prone*. A mistake in a safety-critical application may cost human lives or fail critical missions.

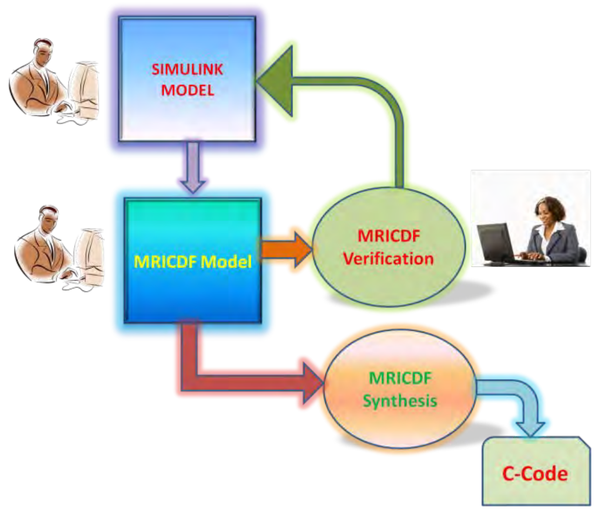
In Europe, research on automated synthesis of safety-critical real-time systems have been carried out over the last two decades [9] resulting in multiple distinct programming models for capturing control algorithms and synthesizing their software implementations. There have been two main programming paradigms – *time driven* and *event driven*. In the time driven paradigm, it is assumed that the reactive system samples or reads inputs on a periodic basis. Programs written in event driven paradigm only reacts when the environment offers one or more input events. It is easier to build a programming model and synthesis algorithms for time-driven control, because the program does not have to decide when to expect the next inputs. Synchronous languages Esterel, Lustre etc., are based on this paradigm. SCADE – a commercial tool -- based on Lustre -- with a visual programming interface -- is used to model many avionics, automotive control software for automated code generation. Air Bus claims that 35% of the code for Air Bus 380 was automatically synthesized.

However, the problems with the time driven paradigm are as follows: (i) due to periodic reading of inputs, and computation of reaction based on presence or absence of inputs, the program must be able to sense *absence* of values on input ports. This is only possible in a time driven synchronous environment. In an asynchronous environment, a perceived absence may be an arbitrary delay of arrival of input. Thus programs synthesized from such modeling languages cannot be directly used in an asynchronous environment, nor can they be arbitrarily composed. This lack of Composability disallows modular development of the software or easy development of distributed software – which is a requisite these days for embedded systems. Thus communicating components (which communicate over buses such as CAN, Flexray, I2C etc.) must all have the same notion of a global time. Therefore, time triggered buses such as CAN bus may be appropriate, but event triggered communication buses will pose challenge. (ii) If the intermittent time between successive events is uneven, the time driven approach requires continuous sampling of inputs at the lowest time scale, leading to inefficiency. The event driven paradigm on the other hand allows components to work in asynchronous environments as it never requires deciding absence. If absence is ever used – it

must be derivable by computation. This allows the ability to compose with both time-triggered and event-triggered buses, and also make them work easily in asynchronous environment of interrupts and other intermittent inputs. The language SIGNAL [9] and its enabling tool set Polychrony implements this programming model. The word *polychronous* means that the notion of time is not globally unique and hence better suited for distributed environments such as an avionics platform like ARINC 653. Moreover, Polychrony is defined in a dataflow programming paradigm rather than imperative, thus allowing one to write programs over streams or flows of events or data, by writing relational constraints between the various asynchronous streams/flows, and thereby allowing a descriptive programming style.

The SIMULINK or Ptolemy II also allow dataflow programming model. However, models built with SIMULINK are used for simulation purpose only. In the recent years, real-time workshop/codegen facility also generates code from SIMULINK models but the caveats are enormous for them to be used in mission-critical applications. First, SIMULINK or Ptolemy II models have no published formal semantics. Some attempts to provide semantics by academics are based on the assumptions about the simulation semantics. Thus the generated code cannot be formally verified against a formal model. One can try to match SIMULINK simulation with the execution of generated code. However, any mistake in simulation (due to ambiguous semantics) is likely to be replicated in the generated code as the same company wrote the simulator and the code generator. Thus code generated this way is neither correct-by-construction, nor can save much development effort because 70% of a project's effort is spent on Validation & Verification (V&V).

In order to provide the DOD with a tool and methodology for embedded safety-critical software synthesis for multi-core embedded platforms, we have been working on a programming model code named **MRICDF** (Multi-Rate Instantaneous Channel Connected Data Flow Network), and the corresponding synthesis algorithms targeting multi-threaded C-code which can be cross compiled into various platforms. As explained already, this programming model is Polychronous, thus exploiting the advantages of asynchronous parallelism inherent in most computational dataflow required in embedded systems such as signal processing, image processing, automated target recognition, automated vision in drones, as well as control applications such as avionics, automotive and weapons control. However, our interaction with various DOD vendors led us to believe that while our programming model is amenable to code synthesis (sequential and multi-threaded) and is supported by formal semantics, refinement based synthesis methodology, and a science of programming that has been developed over two decades in Europe, and further developed by us; the embedded system designers do not always accept a new programming paradigm that they are not trained in. Therefore, we also provide a SIMULINK front-end, so that the engineers can design their software specification with a SIMULINK front-end with all the programming elements and libraries of SIMULINK. We will define semantics of SIMULINK in our polychronous programming model, so that the model designed by engineers has a formal semantics in the form of MRICDF networks. Therefore, we can use our formal verification tools that work on MRICDF models to check for functional correctness, and consistency, and also can synthesize multi-threaded C-code.



SIMULINK to MRICDF – AUTOMATED (**this project**)

MRICDF verification – AUTOMATED – (**related project**)

Feedback after Verification – AUTOMATED -- (**related project**)

C-Code Synthesis from MRICDF – AUTOMATED (**this project**)

Furthermore, our goal in this project has not only been to develop the fundamental theory, and algorithms, and prototype tools, but to initiate planned technology transfer to a DOD vendor company, which can then further develop the tools, and technology that can be made DOD ready as requisitioned by DOD. We have thus partnered with ***L-3 Communications***, an experienced DOD vendor with CMM-3 level software process maturity. We transferred our methodology, design, code, and provided sufficient training so that they could create a DOD ready tool and methodology for multi-threaded C-code synthesis from SIMULINK models via MRICDF intermediate model. This tool and methodology can then be used from DOD embedded systems vendors for creating provably correct multi-threaded embedded software. The advantages of this methodology are that (i) the verification burden will be greatly reduced (currently it is 70% of a software design cycle but can be reduced to 30% or so because the verification will now be done at the intermediate formal model in the form of MRICDF with formal verification tool that will be produced as a part of another project); (ii) the code generated is provably correct provided the synthesis tool is correctly implemented. Proving the correctness of the synthesis tool formally is beyond the scope of this project but could be undertaken as a different project.

Clearly, the budget and time span for this project has not permitted the completion of a certifiable tool to be used right away, but the initial technology transfer to a DOD vendor has been accomplished, which hopefully has established the base line implementation which can then be customized as requirements arise in various DOD domains such as in F-35 follow-on projects in Lockheed, and projects at other DOD vendors which require guaranteed correctness of code along with real-time requirements, and wants to enhance productivity via formal modeling and synthesis driven methodology and tool set.

1 Introduction

Embedded systems are omnipresent and have become an integral part of our day-to-day lives. The car we drive, the plane we travel in, the entertainment center at home and the smart-phones we use for communication – all have embedded systems in them. Some of these embedded systems are complex and some of these complex embedded systems are also safety-critical. Examples of safety critical embedded systems are found in automobiles, avionics, nuclear plants, etc. Development of complex safety-critical embedded systems require design methodologies that not only manage the complexity, but also provide guarantees about correctness of the system. These methodologies cannot be extensions of traditional approaches that are tailored for development of hardware or development of software alone. Also, by following traditional methodologies, providing guarantees of correctness for the system usually requires exhaustive testing, which might not be possible for complex systems. Formal model-driven design approaches provide an alternate approach for developing such systems. And further, by using various formal tools one can verify properties of the system being modeled and can reason about it's correctness. In this report, we have explored a "correct-by-construction" methodology for developing safety-critical embedded systems. We have adopted a formal modeling language MRICDF - (Multi-rate Instantaneous Channel Connected Data-flow), and investigated model-driven techniques for design and development of safety-critical embedded systems. Our investigations are spread across three areas, and are explained in the following sections.

1.1Software Synthesis

Over the past two decades, the embedded computing world has increasingly preferred software over hardware for implementation of functionalities. One of the major advantages that software offers over hardware is – higher flexibility for developers to rapidly adapt last minute changes in user requirements, without paying the high re-development cost associated with changing the hardware. As a result of this preference, the amount of software in the modern embedded systems is staggering. Figure 1 shows the deployed volume of software in selected embedded systems annually.

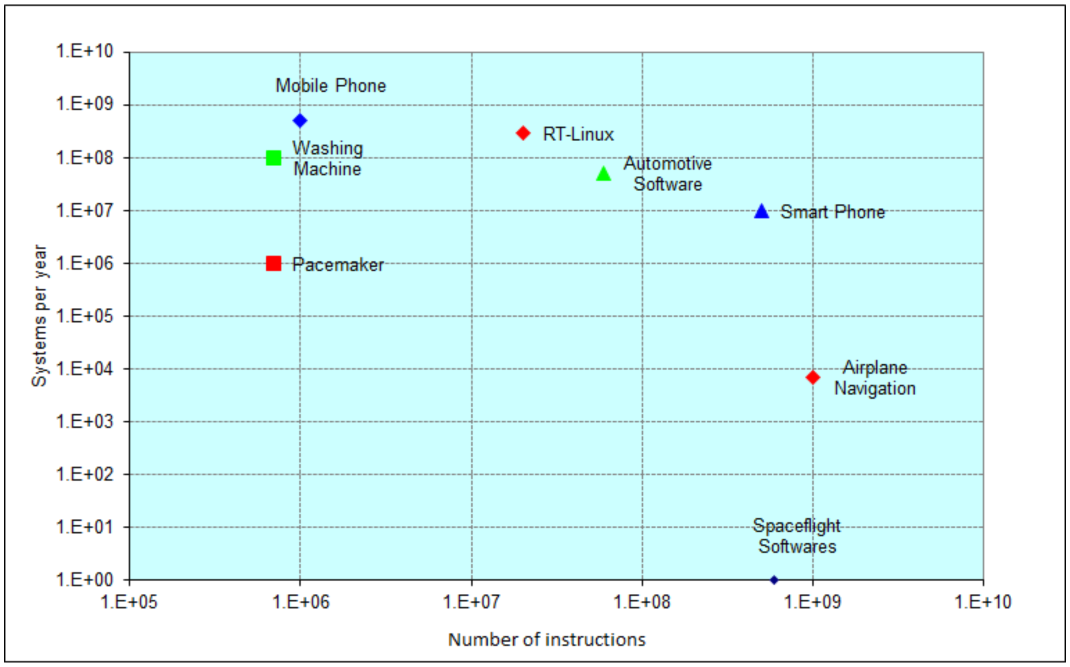


Figure 1: Deployed Volume of Software in Selected Embedded Systems Annually [23]

The trend of "increasing use of software" can be even seen in safety-critical embedded systems, especially the ones used in avionics, automotive and medical devices. Figure 2 shows the amount of software used in selected safety critical embedded systems over the last few decades. Developers of safety-critical embedded systems are not just adopting software to implement non-critical system functions, but also to implement critical control system functions. With the complexity of these safety critical systems increasing at an exponential rate, the size and the complexity of software used to control them are also increasing.

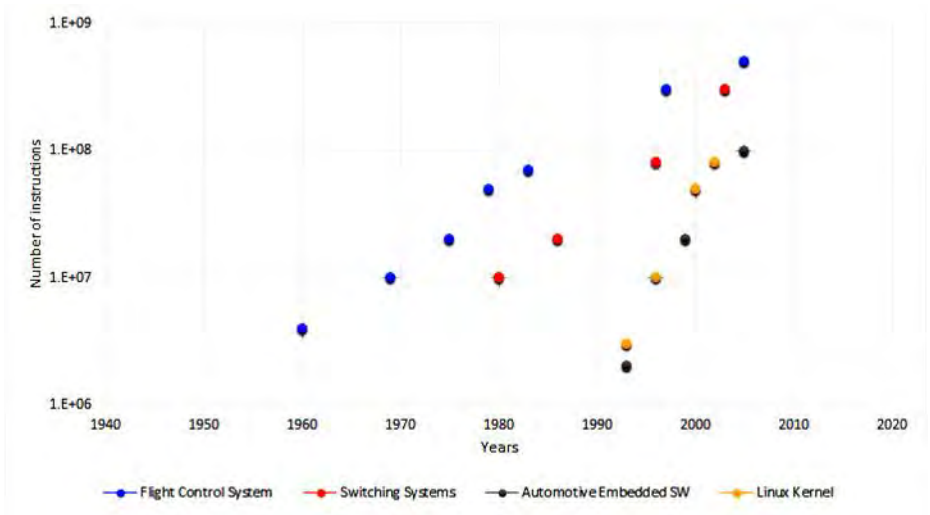


Figure 2: Amount of Software Used in Safety Critical Systems Over Years [23]

Developing error-free software for such complex systems and providing correctness guarantees by following traditional software design approaches can be very difficult. Formal correct-by-construction techniques provide alternative approaches to not only develop safety-critical software, but also to reason about their correctness. These techniques usually require the user to express the functionality of the required software as specifications using a formal language. Some of the formal languages are graphical, while others are textual. Formal tools then apply correctness preserving transformations and synthesize software from these specifications. Further, these tools are also equipped with techniques to prove properties of the specifications and reason about correctness. As the tools use correctness preserving transformations, the properties proved on the specifications are also valid for the synthesized software. Apart from formal languages and tools, one can also use non-formal modeling tools such as Simulink [83] or semi-formal modeling tools such as Unified Modeling Language (UML) [85] for synthesis purposes, though it can be difficult to provide correctness guarantees.

Formal modeling languages are endowed with mathematically well defined semantics but the choice of a formal language for the purpose of software synthesis depends on various factors. Some of the important ones are as follows:

- Model of Computation: Depending on the system to be modeled and the type of code to be synthesized, one can choose a specific model of computation among various available ones. Examples are as follows,
    - Synchronous model of computation: Assumption of apriori external synchronization.
    - Polychronous model of computation: No global notion of time and hence no apriori assumption of unnecessary synchronization - synchronization only as needed.

- Ease of modeling: A perceived quality that indicates how easy or difficult it is to a model complex system using the language.
- Tools and Techniques: Availability of tools that analyze models in a particular language and their ability to do synthesis, verification and validation.
- Ease of learning: Harder the language to learn, less likely it is to be adopted in the industry.

In the past, many research projects have tackled the problem of synthesis of sequential code from formal specifications. We have targeted our research on synthesizing of concurrent code from formal polychronous specifications. In particular, we have considered a formal modeling language which is polychronous in nature - MRICDF (Multi-Rate Instantaneous Channel-connected Data Flow). The reason for choosing MRICDF polychronous language is explained in detail in Section 2. In Section 3.2, we explain the syntax, semantics and other aspects of MRICDF programming language and elaborate more on the past work of sequential code synthesis from polychronous MRICDF models. In Section 3.3, we explain a novel approach for concurrent software synthesis from polychronous MRICDF models. Excessive synchronizations between threads/tasks in concurrent software can adversely affect its performance. Based on the analysis of affine-relations between clocks of signals, a novel technique explained in Section 3.4 identifies synchronizations that could be removed/avoided without affecting the behavior of the system. When such avoidable synchronizations are removed, the efficiency of the synthesized multi-threaded code improves.

1.2 Hardware Synthesis

Despite all the advantages that software offers, for certain safety-critical applications that are power and performance critical, application specific hardware platforms are still preferred over software running on general purpose microprocessors. Also, developers of large applications, often isolate performance and power critical parts of the large application and offload it on to specific hardware. Application Specific Instruction-set Processors (ASIPs) are ideally suited for such purposes. ASIPs are basically processors with customized instruction sets that are designed to exploit special characteristics in a class of applications. Custom instruction sets of ASIPs allow the designer to maintain a high level of design flexibility, yet offers better performance and consumes smaller area. ASIPs will also allow better reuse of components by doing resource sharing during the different modes of operation. Designing such ASIPs while keeping the area minimum and not sacrificing latency or clock speed is a much researched problem. Figure 3 shows the typical steps in any ASIP design methodology.

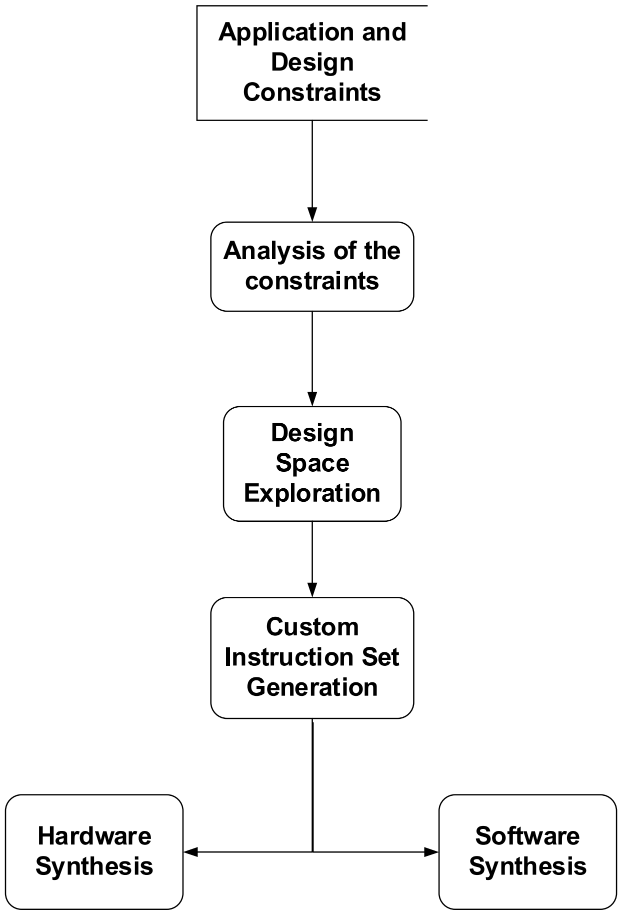

Application and Design Constraints
Analysis of the constraints
Design Space Exploration
Custom Instruction Set Generation
Hardware Synthesis
Software Synthesis

Figure 3: Flow of ASIP Design Methodology [38]

Typically, the design process starts with analysis of the application by the designer. After analysis, the designer performs architectural design space exploration to determine a suitable architecture that satisfies the power and performance requirements. To do this, a good parametrized abstract model of the application is essential. The designer then, manually decides on the structure of the hardware and expresses it in hardware description languages (HDLs) such as Verilog or VHDL with behaviors attached to structures. Further, the designer instructs the CAD tools about the frequently occurring patterns and directs them towards synthesizing of instruction sets. Based on this instruction set, software for the application is then developed.

As the complexity of the design increases, designer's task becomes increasingly complicated and optimality is much harder to achieve by using the traditional CAD (Computer Aided Design) tools. An alternate approach is to formally model the application and apply transformations on the model to determine optimal architectural solutions, for resource estimation and identification of sharable resources. Conditional Partial Order Graphs (CPOGs), a graph-based abstraction model provides a compact and efficient way to formally represent the operation of an application. Further, the abstracted model can be subjected to transformations to synthesize custom instruction sets. But, modeling the software part in some language and the hardware in some other language will require tools that are capable of handling both the languages and reason about the correctness of the entire system. It would be better, if the same formal language that was adopted during synthesis of software, can also be adopted for specifying hardware. This would enable co-specification and co-synthesis of hardware and software. This would also allow reuse of the existing verification and validation tools.

In our research, we have explored the problem of synthesis of Application Specific Instruction-Set Processors (ASIPs) from formal polychronous specifications by converting it to CPOGs first and then using the CPOGs to synthesize ASIPs. This has been described in detail in Section 3.5.

1.3 Verification and Validation

Formal synthesis techniques usually employ correctness preserving transformations to convert the initial specifications/models to target software and/or hardware. This means that, if there are any errors or bugs in the initial specifications/models that weren't fixed, then the correctness preserving transformations will propagate these errors all the way to the end result (synthesized software and/or hardware). Thus, it is important to validate the initial specifications/models before applying the transformations. Formal verification tools such as model checkers, theorem provers, SAT (Satisfiability) solvers, SMT (Satisfiability Modulo Theories) solvers, Linear and Non-linear Polyhedral frameworks, etc., can be employed for such validation purposes. For example, if model checking technique is adopted, then the model and the desired properties of the model are formulated and a model checking tool can automatically check if the given model satisfies the desired property or not. Selecting the model checking tool depends on various factors such as – the type of constraints in the model, the property to be verified, computational complexity of the algorithm implemented in the tool, etc. If the model fails to satisfy the property, which happens more often than not, the designers will be interested either obtaining a single constraint under which the property fails or obtaining a complete set of constraints on the model where the property always fails. SAT and SMT solvers are highly efficient at providing single counter examples, while all-sat tools or polyhedral analysis tools are better are providing set of constraints under which property fails.

For successful synthesis of software and/or hardware and consistent running of the synthesized software and/or hardware, there are many properties that a model has to satisfy. Among them, we consider three prominent properties and are listed below -

• *Causality* - the model under consideration should not contain any constructive causal loops

• *Dimensional Consistency* - the units/dimensions of the signals at the interfaces should be consistent

• *Value Range Consistency* - the range of values a signal can take does not violate any pre-specified/assumed user constraints

1.3.1 Causality Analysis

During modeling, designers might unknowingly introduce causal loops into the models. These causal loops might be *constructive* or *non-constructive*. A simple graph-based dependency analysis should be able to identify all the causal loops and some more. But, this approach, though sound, might yield a lot of false positives, which will result in models with *apparent/non-constructive* causal loops being mistakenly rejected. We show that, to reduce false positives and improve the causality analysis, SAT and SMT solvers can be very effective. These solvers differentiate between *true* and *false* causal loops by looking for any contradicting dependency conditions along the loop. In the first part of Section 3.6, we explain in detail the idea of using SAT/SMT solvers to improve causal analysis techniques.

As a designer, apart from knowing that a causal loop is *constructive*, it would be also very helpful to know the physical input constraints under which the causal loop is *constructive*. If the designer can ensure that, those combination of physical inputs will never occur in a realistic system, then, despite the causal loop being *constructive*, it is never *realizable* – and hence the model can be accepted. In the latter part of Section 3.6, we propose a novel polyhedral model-based analysis technique which identifies constructive causal loops and also provides the input constraints under which the causal loop is realizable. Further in the same section, we propose a wrapper generation technique which prevents these unwanted inputs to the system by filtering them and help operate the system in a safe operating region.

1.3.2 Type Consistency Analysis

Embedded software often interacts with the physical world through signals that represent physical quantities such as velocity, power, etc. These physical quantities are characterized by dimensions and units. But seldom we embed any of these domain specific information into the software. As a result of this, generic type checking done on the software can only ensure identification of mismatched data types between the connected software components. It does not check for dimensional and unit inconsistencies. Extending the generated software with type and unit information for analysis, will not only require a change in coding language and compiler, but also will result in additional runtime overhead. Alternatively, if one is using model-based correct-by-construction approach for software development, then performing the type analysis statically at the model level and ensuring correctness can not only enhance correctness of the final implementation but also save on costs of fixing a bug found in field testing. In Section 3.7, we present a framework in which users can specify domain specific information and perform type inference and clock calculus-based type consistency analysis on the polychronous MRICDF models.

1.3.3 Value Range Analysis

Safety-critical applications require software and/or hardware that always produces deterministic and correct outputs. Ensuring that a software produces only expected outputs can be done either by doing an exhaustive simulation (non-scalable) and or by static analysis techniques. “Value range analysis” techniques refers to a genre of analysis techniques that statically estimate the range of values a variable can assume during the program execution. Often, such analysis techniques involves a difficult and hard to automate abstraction step. With MRICDF model-based design approach, we can avoid the abstraction step completely and perform range analysis directly on the MRICDF model. Given input range constraints, in Section 3.8, we present rules to derive the range constraints of the outputs. Further in the same section, we propose an approach that exports the range constraints of a model as SMT constraints and show how they can be used to verify various properties related to value ranges of signals.

1.4 Summary of contributions

In this report, we have mainly explored formal model driven techniques for,

- Multi-threaded code synthesis from MRICDF models;
- Application Specific Instruction Set Processor (ASIP) synthesis from MRICDF models;
- Causality Analysis of MRICDF models;
- Units and Dimensional Analysis of MRICDF models;
- Value range analysis of MRICDF models.

We have prototyped all these techniques in our open source tool - EmCodeSyn. We now briefly summarize each of these contributions. For a detail explanation, we refer the reader to further sections of the report.

1.4.1 Contributions towards Multi-threaded code synthesis from MRICDF Models : (Sections 3.3 and 3.4)

Previous attempts at code synthesis from MRICDF models ([43], [40]) were specifically targeted at sequential code synthesis. We concentrated our research efforts towards synthesis of multi-threaded code from MRICDF models. We proposed a novel Boolean theory-based approach for determining if a given MRICDF model is concurrently implementable or not. Furthermore, our Boolean theory involves generation of prime implicates using SMT solvers. We proposed a notion of partial triggers and proposed a technique to infer partial triggers from the prime implicates. Further, we proposed technique to identify the synchronization constraints between the partial triggers. We then proposed a code generation technique by mapping the partial triggers to threads. We performed scalability and performance analysis of the proposed technique. For the considered benchmarks, we noticed that the performance of the synthesized multi-threaded code was about 18% slower than the performance of the hand-written multi-threaded code. Performance analysis revealed a few bottle necks that was causing the dip in the performance of the synthesized multi-threaded code. One of them was - excessive synchronizations. We proposed a novel technique, based on analysis of affine clocks that identifies all the avoidable synchronizations and removes them from the synthesized code, which in-turn improves the performance of the multi-threaded code.

1.4.2 Contributions towards Application Specific Instruction Set Processor (ASIP) synthesis from MRICDF Models : (Section 3.5)

In [55], the authors explained how Conditional Partial Order Graphs (CPOGs) enable us to compactly and efficiently describe and store instruction sets. Further, they explained how they can be used to identify parallelisms and synthesize custom instruction sets. On the same line of thought, we proposed a technique that accepts formal MRICDF/SIGNAL [43] specifications and compiles them to Conditional Partial Order Graphs (CPOGs). These CPOGs are further used to generate custom instruction sets for Application Specific Instruction set Processors (ASIPs).

1.4.3 Contributions towards Causality Analysis of MRICDF Models : (Section 3.6)

One of the pre-requisites for an MRICDF model to be sequentially or concurrently implementable is that it should not contain any causal loops. In the past, numerous solutions have been proposed for doing causality analysis. However, most of these approaches only work on Boolean abstraction of the predicates. This may lead to sound but imprecise decisions being made, which in-turn may lead to erroneously rejecting an MRICDF model to be non-synthesizable. We proposed an SMT and Polyhedra-based approach for performing causality analysis which considers both Boolean and Integer predicates. Our proposed approach helps in making better decisions while performing causal analysis. Furthermore, we proposed an approach to identify the constraints under which the causality behavior of the system is exhibited. Then, we explained how these constraints can be used to generate a wrapper which would always keep the system in safe operating region.

1.4.4 Contributions towards Units and Dimensional Analysis of MRICDF Models : (Section 3.7)

Units and dimensional inconsistencies between signals at the interfaces could result in catastrophic failures. We proposed an novel SMT-based approach for performing unit and dimensional analysis statically on the polychronous models. To the best of our knowledge, this is the first ever approach for performing dimensional analysis on polychronous languages. The main advantage of our approach is that it considers the clock constraints of the signals which checking for dimensional consistencies. Our approach is scalable and adds minimum overhead.

1.4.5 Contributions towards Value Range Analysis of MRICDF Models : (Section 3.8)

Software used in safety critical embedded system is required to produce expected output values for every possible run. By conducting static value range analysis on the program, one can check if the signals ever take any values out of some pre-defined bound. There are approaches proposed in the past for doing value range analysis for synchronous programming languages such as C/C++/Java. The polychronous model of computation brings in additional complexity which would require the value range analysis techniques to consider the clocks of the signals along with their values. We proposed a novel SMT-based technique to perform value range analysis in polychronous languages and explained it with a case study. Our proposed approach considers the clocks of the signals too.

1.5 Organization of the Report

This report is organized around three main topics, namely – *(a) Software Synthesis, (b) Hardware Synthesis, and (c) Verification & Validation*, as shown in Figure 4.

Section 2 discusses various models of computation, abstractions of time, formalisms associated with popular synchronous and polychronous languages such as Esterel [12], Quartz [68], Lustre [34], Signal [32], etc.

Section 3.1 initially discusses our results related to multi-threaded code synthesis and high level synthesis from formal languages. Further, it discusses one work related to property verification – causality detection in particular and type system extension and type checking in modeling languages.

Section 3.2 articulates the formalisms, syntax and semantics associated with the MRICDF polychronous modeling language. Further in the section, the tool EmCodeSyn that accepts and analyses MRICDF specifications is introduced. A short description of its functionalities and the usage flows for various operations such as code generation, type checking, etc. is provided. The last part of the section describes a Boolean theory-based sequential software synthesis approach.

The first part of Section 3.3 describes a Boolean theory-based approach to identify concurrent implementability of MRICDF models. The second part of the section describes the algorithms and implementation details of the proposed approach along with experimental results. The Appendix lists an example MRICDF model and the corresponding multi-threaded code generated for this model using the EmCodeSyn tool.

In Section 3.4, we explore techniques to improve the efficiency of the synthesized multi-threaded code by identifying avoidable synchronizations using affine relations between the clocks of signals.

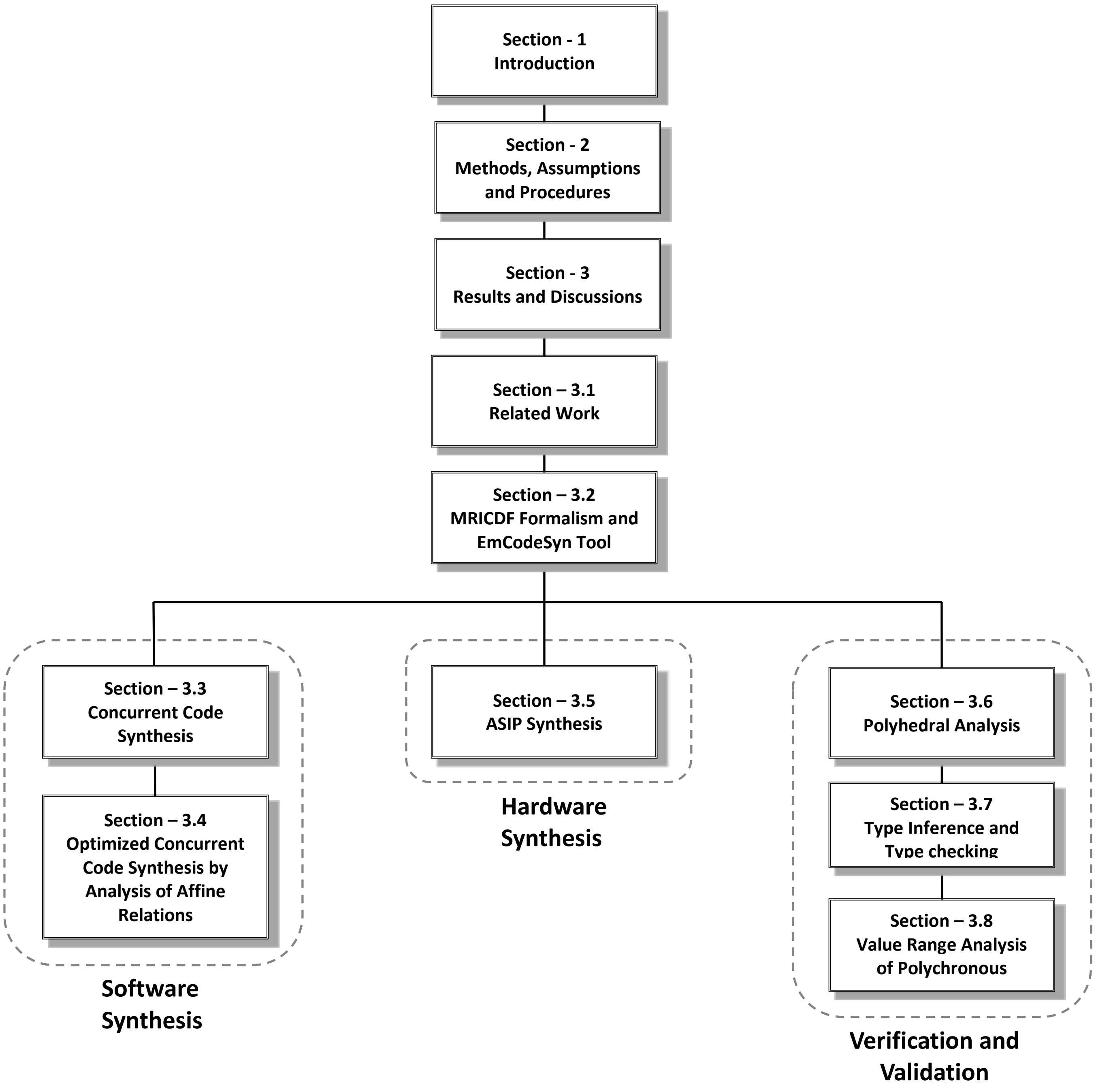
Section 3.5 outlines our proposed approach to convert MRICDF models to Conditional Partial Order Graphs(CPOGs), which are further utilized to synthesize ASIPs.

Section 3.6 describes the use of polyhedral analysis for verifying properties of polychronous specifications, with a focus on verifying the presence or absence of causal loops. Explained further in the section is a wrapper generation technique, that utilizes the bounds of safe operating area obtained by the polyhedral analysis and generates wrappers which filters the unsafe inputs to the system.

Section 3.7 describes the first attempt at type inference for polychronous specifications that includes the clocks of signals. It also explains a fully automated SMT-based type consistency checking approach.

In Section 3.8, we propose another novel approach to derive the range constraints of signals in polychronous models. Further, we show how to export the range constraints as SMT constraints and explain how they can be used to verify properties related to value ranges in polychronous models.

Section 4 discusses the conclusions of the above research works and proposes ideas for future work.



Section - 1
Introduction
Section - 2
Methods, Assumptions and Procedures
Section - 3
Results and Discussions
Section – 3.1
Related Work
Section – 3.2
MRICDF Formalism and EmCodeSyn Tool
Section – 3.3
Concurrent Code Synthesis
Section – 3.4
Optimized Concurrent Code Synthesis by Analysis of Affine Relations
Software Synthesis
Section – 3.5
ASIP Synthesis
Hardware Synthesis
Section – 3.6
Polyhedral Analysis
Section – 3.7
Type Inference and Type checking
Section – 3.8
Value Range Analysis of Polychronous
Verification and Validation

Figure 4: Report Organization

1.6 Publications on the work reported in this report

The following are the peer-reviewed publications on the work done in this report. All the work in terms of the research contributions, implementation and experimentation for these publications was done by the author under the guidance of Dr. Sandeep K. Shukla.

1. **M Nanjundappa**, M Kracht, J Ouy and SK Shukla, *Synthesizing embedded software with safety wrappers through polyhedral analysis in a polychronous framework* – IEEE Electronic System Level Synthesis Conference (ESLSyn), 2012

2. **M Nanjundappa**, MW Kracht, J Ouy and SK Shukla, *A New Multi-threaded Code Synthesis Methodology and Tool for Correct-by- Construction Synthesis from Polychronous Specifications* – ACM International Conference on Application of Concurrency to System Design (ACSD), 2013

3. **M Nanjundappa** and SK Shukla, *Compiling polychronous programs into conditional partial orders for ASIP synthesis* – ACM Formal Methods in Software Engineering (FormaliSE), 2014

4. **M Nanjundappa** and SK Shukla, *Verification of Unit and Dimensional Consistencies in Polychronous Specifications* – IEEE Forum on specification & Design Languages (FDL), 2014

5. MW Kracht, **M Nanjundappa** and SK Shukla, *Modeling and Scheduling of Multi-Periodic Real-Time Tasks with Conditional Behaviors using Polychronous Specifications*

6. **M Nanjundappa** and SK Shukla, *SMT based value range analysis of Polychronous Models*

7. **M Nanjundappa** and SK Shukla, *Sythesis of Improved Multi-threaded code from Polychronous specifications using Affine clock relations*

8. **M Nanjundappa**, BA Jose, HD Patel and SK Shukla, *SCGPSim: A fast SystemC simulator on GPUs* – IEEE 15th Asia and South Pacific Design Automation Conference (ASP-DAC), 2010 – **Best Paper Award**

9. **M Nanjundappa**, A Kaushik, HD Patel and SK Shukla, *Accelerating SystemC simulations using GPUs* – IEEE International High Level Design Validation and Test (HLDVT) Workshop, 2012

2 Methods, Assumptions and Procedures

Past three decades have seen extensive research being done in the area of automated synthesis of safety critical embedded control software. The common goal of all these research efforts is – *to automatically synthesize correct and efficient code from high-level descriptions of the system*. Initial research efforts were targeted towards synthesis of sequential or single-threaded code, while most of the recent efforts are targeted towards multi-threaded code synthesis – thanks to increasing computation demands and increasing popularity of multi-core embedded processors. The high-level description of the desired system is usually described using a modeling language. The semantics of these languages can be either formal, semi-formal or non-formal. Depending on the properties of the system to be modeled, the designer can choose a modeling language that offers the desired model of computation. Further, these descriptions are analyzed and transformed into software (C/C++ code) or hardware (Register-Transfer Level (RTL) implementations) by a tool. One class of modeling languages that are often used for modeling reactive embedded systems are *Synchronous Languages*. Table 1 lists some of the popular *synchronous programming languages* that are used to describe embedded systems.

Table 1: Popular Modeling Languages used for designing Embedded Systems

No.	Modeling Language	Graphical/Textual	Semantics	Highlights
1	Argos	Graphical	Formal	Synchronous, Imperative
2	Atom	Textual	Formal	Synchronous Language
3	Esterel	Textual	Formal	Synchronous, Imperative
4	Lucid Synchrone	Textual	Formal	Synchronous, Declarative
5	Lustre	Textual	Formal	Synchronous, Declarative
6	MRICDF	Graphical	Formal	Polychronous, Declarative
7	Quartx	Textual	Formal	Synchronous, Imperative
8	Reactive-C	Textual	Non-formal	Imperative, Extension to C
9	Signal	Textual	Formal	Polychronous, Declarative
10	Statecharts	Graphical	Formal	Synchronous, Imperative
11	SyncCharts	Graphical	Formal	Synchronous, Imperative

Synchronous languages were introduced as a means to enable deterministic and correct-by-construction development of embedded systems that are safety critical. These languages are equipped with formal semantics, which will allow the designers to unambiguously describe the required behavior of the intended system. The mathematical foundations of these languages allow for extensive and efficient analysis, verification & validation and further provide a sound basis upon which we can reason about the correctness of the system. Synchronous languages rely on *synchrony hypothesis*. The synchrony hypothesis states that, the system is fast enough to respond to the previous set of input events before the next set of input events occur. This indirectly implies that, the execution of a synchronous process can be divided into a discrete set of computation steps (macro-steps). These steps are usually called as logical instants. The synchrony hypothesis requires that within each of these logical instants, the computation is well ordered – in other words, there are no cyclic dependencies between computations (micro-steps) inside each logical instant. The model of computation implemented by synchronous languages is known as Synchronous model of computation.

Polychronous languages are also synchronous languages. They also rely on synchrony hypothesis, but extend it to process multiple clocks. This means that, there can be several synchronous processes which are running asynchronously until some communication occurs; within each synchronous process and between communicating synchronous processes, synchrony hypothesis is assumed. The model of computation implemented by polychronous languages is known as Polychronous model of computation. These languages are typically used to model an important class of systems that exhibit globally asynchronous but locally synchronous behavior.

Most of the formal languages listed in Table 1 are examples of **Synchronous Languages**. MRICDF and Signal are examples of **Polychronous Languages**.

2.1 Synchronous and Polychronous Model of Computation

The synchronous model of computation enables the designer to describe systems that are driven by a single global clock – mono-clocked systems, while the polychronous model of computation allows us to describe systems that are driven by a set of independent clocks – multi-clocked systems. In mono-clocked systems, this global clock is also known as *master clock*. Master clock or its derivatives initiates all the reactions of each subsystem in the mono-clocked system. This implies that the set of reactions initiated by some derivative clock of master clock is a subset of the set of reactions initiated by the master clock itself. Block diagram of a generic mono-clocked system is shown in Figure 5. The rate at which Subsystem A has activations is ten times less than that of rate at which Subsystem B has reactions, while the rate at which Subsystem C has reactions is three times less than that of Subsystem B. Thus, the set of execution instants when subsystem A and C has reactions is a subset of set of execution instants when subsystem B has reactions. This is because as the global clock is same as that of clock of subsystem B. If the behavior of clock of global clock is modified, then the behaviors of clocks of other subsystems also will be modified. Thus, there exists a relation between clocks of all the subsystems and the global clock.

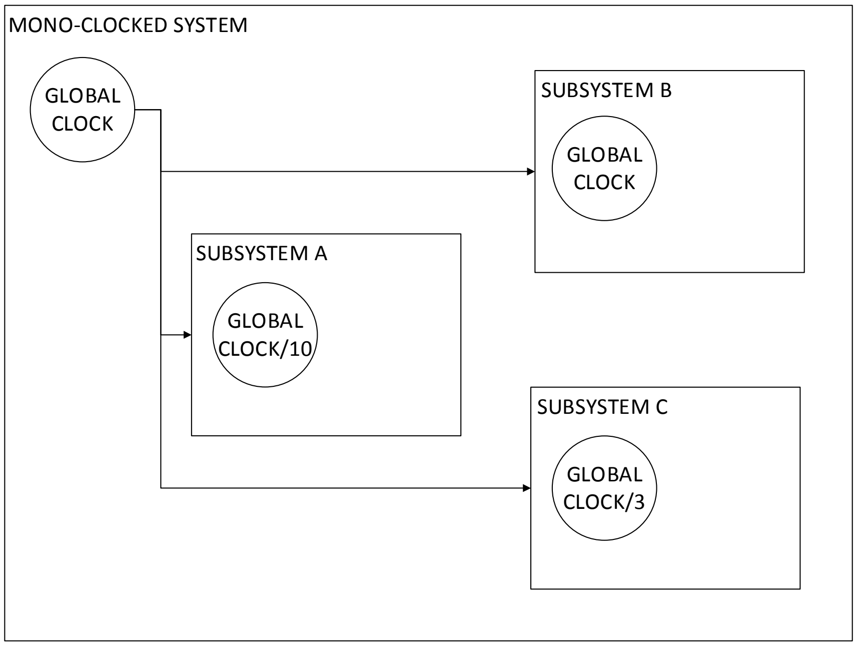


Figure 5: Block Diagram of a Mono-clocked System

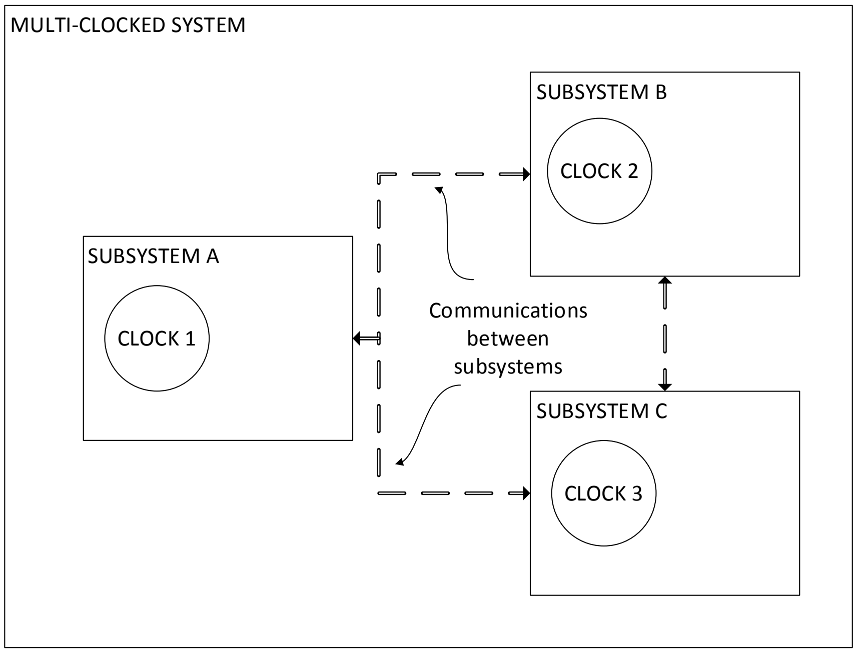


Figure 6: Block Diagram of a Multi-clocked System

Figure 6 shows the block diagram of a generic multi-clocked system. It can be seen each subsystem has its own clock and there is no global clock. Since, the clocks of each subsystem is independent of other clocks, designing of a multi-clocked system can be done in a modular way and composed at the end. When the subsystems need to communicate to each other to sychronize or exchange data, they use shared resources. In future, even if a single subsystem is modified locally, it does not affect the behavior of other subsystems. This feature is highly desired during development of complex embedded systems.

In Section 2.2, we discuss some of the popular synchronous and polychronous languages.

2.2Synchronous Languages

Based on the programming style, synchronous languages can be grouped as *imperative* and *declarative* languages. While modeling in imperative languages, the user explicitly describes the sequence of execution steps, where as while modeling in declarative languages, the user expresses only functional or relational dependencies.

2.2.1 Esterel

The Esterel language [12] is an imperative synchronous language, that can be used to describe complex reactive systems and synthesize both C code and RTL implementations for the same. Esterel language [12] has two basic types of objects – signals and variables. Signals are the basic means of communications. They can be used to represent inputs and outputs of a process or they can be used as local signals inside the process. A signal has two attributes associated with it: *status* and *value*. The *status* attribute indicates whether a signal is present or absent in the given logical instant and the *value* attribute indicates the data that the signal contains. The *value* attribute is permanent, which means that, if a signal is absent in the current logical instant, it will retain the value from the previous instant when the signal was last present. The set of logical instants where the signal is present is termed as its *logical clock*. Each Esterel program has a predefined signal usually referred to as *tick*. The clock of the signal *tick*, is termed as global clock and is faster than all the other logical clocks. This implies that, the set of logical instants where all other clocks are present is a subset of the set of logical instants where tick is present.

An Esterel program is made up of *modules*, which is in turn made up of *interface* – that represents input and output signals, and the *body* – that represents the behavior of the module. The body consists of imperative and reactive statements that are made up from the basic statements shown in Table 2. Whenever the activation signal corresponding to a module is activated, then the body of the model is executed instanteneously. Listing 2.1 shows an example Esterel program that keeps track of number of people in a room and notifies that room is FULL when there are 3 people inside and EMPTY when there is no one inside the room.

Table 2: Basic Esterel Statements [12]

Statement	Explanation
emit s	Immediately signal s is made present
present s then T1 else T2 end	If the signal s is present, do T1 else do T2
Pause	Pauses the execution of the current thread until next reaction occurs
T1; T2	First perform T1 and then perform T2
loop T1 end	Keep repeating T1 forever

loop T1 each s	Keep repeating T1 whenever s occurs; if s occurs while T1 is in progress, then stop T1 immediately and restart it again
await s	Wait until s occurs
T1 || T2	Start T1 and T2 together
abort T1 when s	Run T1 (i) until T1 is finished, or (ii) until a reaction when s is present and T1 is not yet complete
suspend T1 when s	Perform T1 except when s is present
sustain s	Keep emitting s; can also be written as loop emit s; pause end
run M	Start running code for module M

Listing 2.1 Example Esterel Program

```
module Counter:
input ADD, SUB;
output FULL, EMPTY;
        var count := 0 : integer in
                loop
                        present ADD then if count < 3 then
                                count := count + 1 end end;
                        present SUB then if count > 0 then
                                count := count - 1 end end;
                        if count = 0 then emit EMPTY end;
                        if count = 3 then emit FULL end;
                        pause
                end
        end
end module
```

Esterel programs are compiled using one of the many available Esterel Compilers such as Esterel INRIA compiler [37], Columbia Esterel Compiler [25], Esterel Studio [77] (commercial tool), etc. We consider here the compiler distributed by INRIA [37]. This compiler initially compiles the Esterel program into finite state automaton with the statements as data-paths and conditions as guards. Further, this finite state machine is used as underlying formal model and is subjected to correctness preserving transformations to synthesize C code or RTL implementations. One can also analyze the inherent concurrency present in the Esterel programs to generate concurrent code. In Section 3.1 we provide an overview of such efforts.

2.2.2 Quartz

Quartz [68] is an imperative synchronous programming language that is based on Esterel. It's developed as a part of Averest project [5] and the Quartz compiler also has the same name – Averest. Quartz extends the Esterel language with statements that can explicitly express non-determinism. This allows Quartz to model distributed systems that do not exhibit synchronous behavior during execution of threads, but instead exhibit asynchronous behavior. Apart from this, Quartz also adds next() statement that allows delayed data assignments and halt statement. The halt statement of Quartz can be written using Esterel statements as loop pause end. Another important advantage that Quartz language has over Esterel language is, its ability to handle analog data [69], which will allow it to be used in designing hybrid systems. A sample example of Quartz program that computes square root of a number is shown in Listing 2.2. The Quartz program is very similar to an Esterel program except the additonal statement next and the drivenby part which is used to provide simulation inputs to the Averest tool.

Listing 2.2 Example Quartz Program

```
macro N = 200;

module SquareRoot(nat ?a,x,event !rdy) {
        nat x_old;
        x = a;
        do {
                next(x_old) = x;
                next(x) = (x+(a/x))/2;
                pause;
        } while(x_old>x);
        emit(rdy);
}
drivenby {
        a = 2*exp(10,N);
        await(rdy);
}
```

2.2.3 Lustre

Lustre [34] is a declarative synchronous programming language developed by Verimag. It is based on data-flow model and is highly suitable for modeling of reactive systems that manipulate dataflows. The commercial version of Lustre and its compiler – SCADE (Safety Critical Application Development Environment) developed by Esterel Technologies, has been adopted by various industries in developing real world safety critical applications [77]. Lustre language has two basic objects: *variables* and *nodes*. Variable represent an infinite flow of values. Similar to the *signal* concept of Esterel, a flow in Lustre is characterized by two attributes: *clock* and *value*. The set of logical instants where the a new value occurs on the variable is called its *clock*. A *clock*, thus can be encoded as a Boolean signal where a *true* value on the encoding signal indicates the presence of the associated flow, and *false* value indicates an absense. *Value* attribute denotes the data contained in the flow. Each *node* represents a programming unit and it is composed of an interface with input/output flows, and a body defined as a set of equations. Lustre program is made up of data flow equations using four basic temporal operators shown in Table 3. Apart from the equations, Lustre program might also consist of assertions. They are used to specify properties of the design. An example Lustre program for a simple timer with reset that outputs an alarm signal is shown in Listing 2.3.

Table 3: Basic Lustre Operators

Temporal Operators	Explanation
y = f($x_1,..,x_n$)	f is an instantaneous function on the flows $x_1,..,x_n$
pre(x)	Returns the previous value in the flow of variable x
-> (followed by)	This operator defines the initial values
z = x when y	This samples the flow x with y. The value of x is assigned to z when y occurs and is true
current(x)	This memorizes the last value of x whenever it is present
T1; T2	The nodes T1 and T2 have to be executed in parallel

Listing 2.3 Example Lustre Program

```
node reset_timer (reset:bool) returns (alarm:bool);
        var time: int;
let
        time = 1 -> if reset then
                        1
                else
                        if pre(time) = 10 then
                                1
                        else
                                pre(time) + 1;
                                alarm = (time = 10);
tel
```

The Lustre compiler analyzes the Lustre programs and performs an operation called *clock calculus* to determine the clock hierarchy of the variables. After doing causality analysis, using the clock hierarchy, a finite state automaton (similar to the one built by Esterel compiler) is built which is then used to generate C code or RTL implementations. Some research efforts have also focused on generating synthesizing concurrent code for distributed platforms from Lustre programs. We give a brief overview of these in Section 3.1.

2.2.4 Signal

Signal [33, 32] is a declarative, multi-rate synchronous (polychronous) language developed by IRISA, France. The important difference between Signal and the previously listed synchronous languages is that Signal considers a mathematical model of time, in terms of partial order relations, to describe multi-clocked systems without the necessity of any abstract global clock. This makes it easier to express behaviors of asynchronous systems. A Signal program consists of processes that are made up of *interface* that defines inputs & outputs and *body*. The body consists of statements expressing functional and temporal relationships. These relations are expressed using one of the four primitive operators listed in Table 4. Each signal s in a Signal program is associated with a Boolean signal called *clock* denoted by $\hat{s}$. The clock of a signal defines the rate at which the signal is being updated. Column 2 of Table 4 shows the default clock relations between the signals of the primitive actors. Along with clock dependencies as indicated by the clock relations, there are also data dependencies between the input and output signals of few primitive actors. When the computation of a signal y is dependent on computation of signal x, then we say that y depends on x and is indicated as $x \rightarrow y$. Data dependencies between input and output signals of primitive actors is shown in column 3 of Table 4. A sample Signal program that computes running average of input values is shown in Listing 2.4.

Table 4: Primitive Signal Operators

Actor definition	Clock Relations	Data Dependency Relations
Function $r = a * b$	$\hat{a} = \hat{b} = \hat{r}$	$a \rightarrow r$ $b \rightarrow r$
Delay $y = x \,\$\, n \; init \; v_1..v_n$	$\hat{y} = \hat{x}$	No dependency
Sampler $y = x$ **when** z	$\hat{y} = \hat{x} \wedge \widehat{[z]}$	$x \xrightarrow{[z]} y$
Merge r = a **default** b	$\hat{r} = \hat{a} \vee \hat{b}$	$a \rightarrow r$ $b \xrightarrow{\hat{b}-\hat{a}} r$

Listing 2.4 Example Signal Program

```
function average =
        ( ? integer input; ! real avg;)
        (| sum := sum$ init 0 + input
        | n := n$ init 0 + 1
        | avg := real(sum)/real(n)
        |)
        where
                integer sum, n;
        end;
```

Signal programs are compiled using the Polychrony [81] compiler. Similar to Lustre compiler, the Polychrony compiler also analyzes the clock relations, builds hierarchical clock relation graph and does causality analysis. If the hierarchical clock relation graph turns out to be a *tree* and there are no causal loops, then we say that that particular Signal code exhibits the Endochrony property. Signal programs that are endochronous can be transformed into deterministic sequential C code. There are times when the hierarchical clock relation graph is not a tree, but it can be made a tree by defining and adding additional clocks to the clock relation graph such that it doesn't induce any causal loops. This process is known as Endochronization, and is done when the target is to generate sequential C code. After endochronizing the Signal program, the resultant clock tree can be used to synthesize sequential C code. If the target is to generate multi-threaded code, then we check if the Signal program exhibits weak endochrony [76] property. If yes, then we can synthesize multi-threaded code for the model. An overview of the research efforts that do synthesis from Signal programs is provided in Section 3.1.

2.2.5 Statecharts, SyncCharts, Argos, Reactive-C

Statechart [35] is a visual formalism that is used to describe complex reactive systems. Statecharts are basically extensions of the traditional state-machines with features to describe communication, concurrency and hierarchy. The concurrency extensions allow the user to describe parallel behaviors of system easily. States in Statecharts may be hierarchical and they can be of two types – *and-state*&*or-state*. Hierarchical *and-states* consists of concurrent sub-states that evolve concurrently, while hierarchical *or-states* contain substates that evolve exclusively. Figure 7 shows a Statechart that represents the abstract execution model of an answering machine.

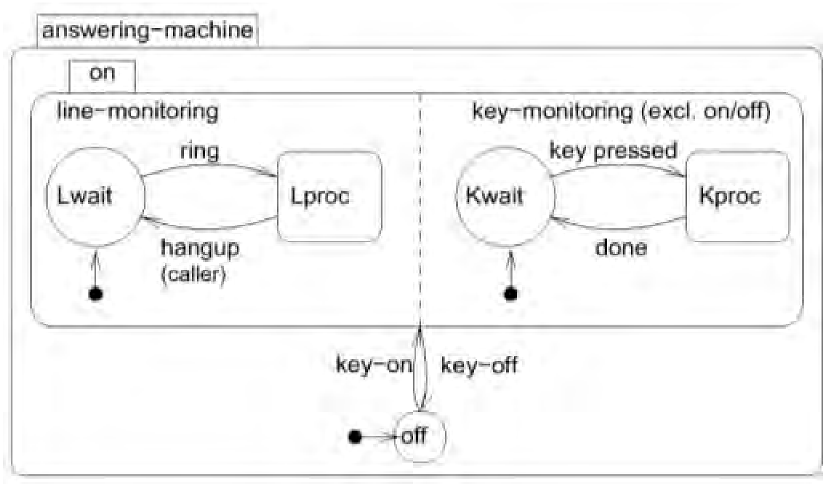


Figure 7: Statechart representing abstract execution model of an answering machine

The unit of reaction in a Statechart is called – *step*. At any given point of time, the current *configuration* of a Statechart is defined by the set of its active states. The transition from current *configuration* to another configuration takes place when a *step* occurs. Tools that interpret Statecharts have different semantics, and based on the semantics considered, the way actions occur at each *step* differs. The tool Statemate interprets Statecharts in a way that the actions that occur in current *step* will only become effective in the next *step*, indicating that the outputs are produced after a temporal delay. Thus, Statemate's interpretation of Statecharts does not result in a strictly synchronous semantics and not necessarily deterministic either.

Argos [52, 53] is a synchronous variant of restricted Statecharts language. One of the restrictions Argos enforces on Statecharts language is that, the actions performed within a step has to become effective in the same step. Argos also eliminates the feature of Statecharts that allows multi-level arrows. This results in a well-defined syntax of Argos programs on which a structural semantics can be based. Based on this structural semantics, Argos also defines compositionality. The synchronous viewpoint of Argos is similar to the one adopted in Esterel.

SyncCharts language [4, 3] is a graphical version of the Esterel language developed by Esterel Technologies. SyncCharts was originally inspired by Statecharts, but it has the semantics of Esterel language. SyncCharts being visual, offer better visual representation of the design, which helps the designer during designing complex systems.

Reactive-C [14] is an extension to C programming language, where the concepts of extensions borrowed from Esterel language. Reactive-C enables a C like programming language to be used to design and develop reactive systems. Listing 2.5 shows an example of a simple sequential *Hello-Bye* program. The keyword rproc indicates the start of definition of reactive procedure. The stop statement is used to define the boundaries of the instants. During execution, if a stop statement is encountered, then it indicates that end of execution for current instant. The next statement after *stop*, will be executed in the next instant. Figure 8 shows the execution flow of the Reactive-C program listed in Listing 2.5.

Listing 2.5 Example Reactive-C Program

```
rproc Seq(){
        exec Hello();
        exec Bye();
}
rproc Hello(){
        printf("hello, world\n");
        stop;
        printf("I repeat: hello, world\n");
}
rproc Bye(){
        stop ;
        printf("Bye!\n");
}
```

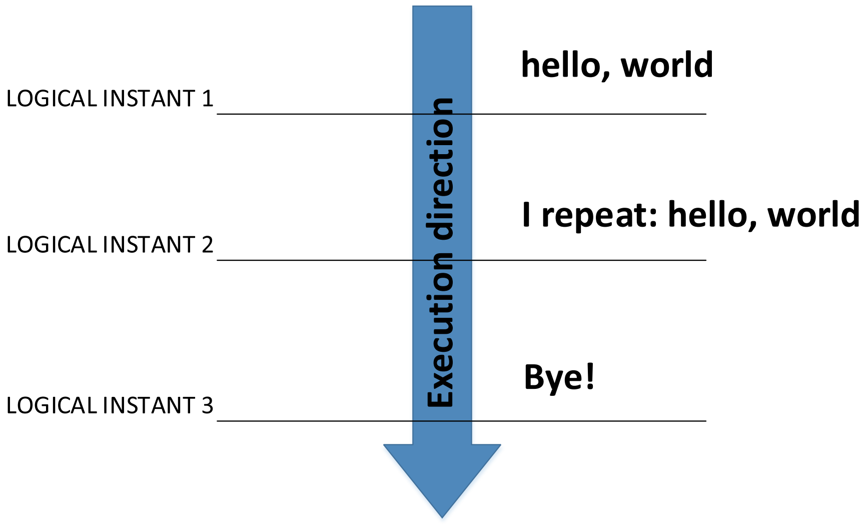


Figure 8: Execution flow of the Reactive-C program shown in Listing 2.5

2.3 Alternatives to Synchronous languages

Apart from synchronous languages listed in Table 1, various other modeling frameworks such as I/O Automata, Kahn Process Networks, Petri-nets, Multi-dimensional SDF, etc. are also used for designing embedded systems. Table 5 lists them.

Table 5: Popular formalisms alternative to Synchronous languages

No.	Modeling Language	Graphical/Textual	Semantics	Highlights
1	Kahn Process Networks (KPN)	Graphical	Formal	Ideally suited to model distributed systems
2	Synchronous Dataflow (SDF) & Multi-dimensional SDF	Graphical	Formal	Ideally suited to model DSP applications
3	Petri Nets	Graphical	Formal	Used for modeling distributed systems
4	I/O Automata	Both	Formal	Used to model asynchronous concurrent systems
5	Ptolemy Framework	Graphical	Formal	Used to model heterogeneous systems, supports variety of MoCs

2.3.1 Kahn Process Networks (KPN)

Kahn Process Networks (KPN) [44, 1] is a specification language proposed by Gilles Kahn for programming distributed systems. In the KPN model of computation, a group of independent sequential processes execute concurrently and communicate with each other in a point-to-point fashion via unbounded First In First Out (FIFO) channels, using a *blocking read* synchronization primitive. A simple example of KPN is shown in Figure 9.

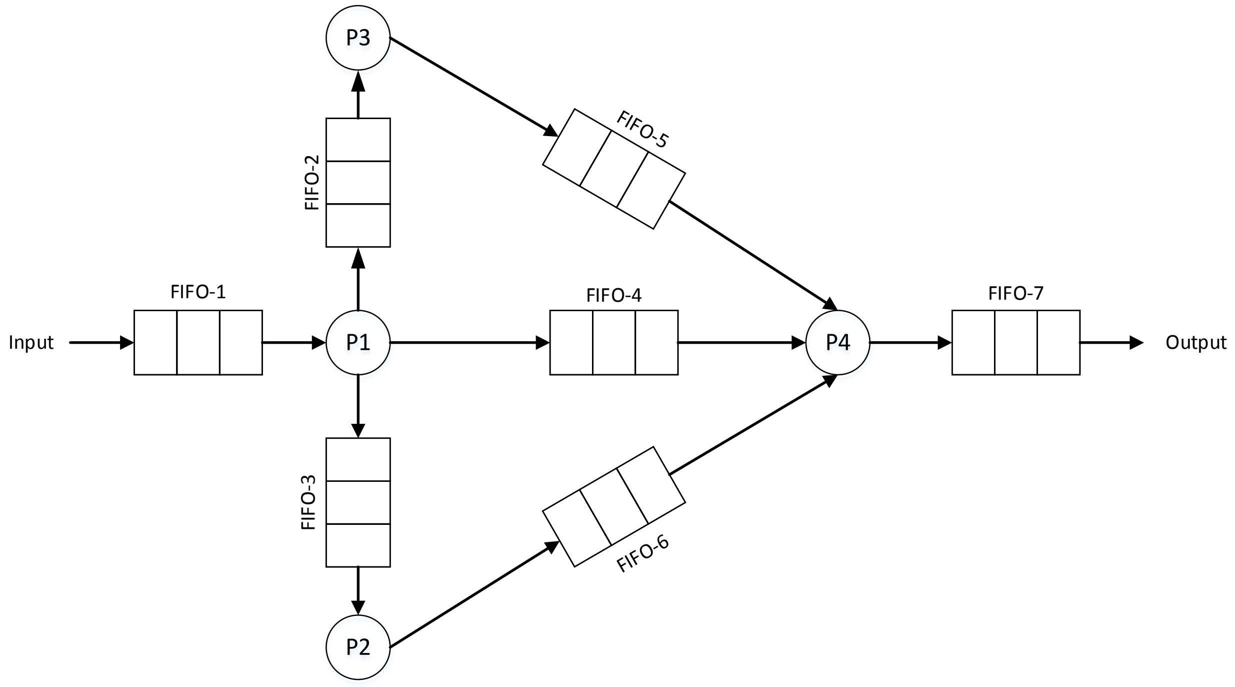


Figure 9: Example of KPN model

Nodes *P1, P2, P3 and P4* represent four independent sequential processes that are running concurrently and communicating via channels represented by the unbounded FIFOs. The channels have the property of *blocking read* and *non-blocking write*. *Blocking read* means that when a FIFO is empty, the process reading that FIFO will be stalled until the FIFO contains enough data tokens. A *non-blocking write* channel indicates that, a process is never stalled during writing and it always succeeds. Some of the features of KPN are as follows,

- KPN model is deterministic - it means irrespective of the schedule, for the same set of inputs, we get same set of outputs
- Simple synchronization primitive - *blocking read*, easy to implement
- Control is completely distributed, no global scheduler
- Communication is point-to-point and is distributed over FIFOs, no global memory concept

All the above features make KPN model of computation an ideal framework to model distributed systems. KPN models can be subjected to various scheduling tools which can estimate the buffer sizes. Further, these KPN models can be easily transformed into Globally Asynchronous, Locally Synchronous (GALS) implementations [75] using correctness preserving transformations, thus making it very attractive for developers who develop safety-critical embedded systems.

2.3.2 Synchronous Dataflow (SDF) and Multi-dimensional (MD-SDF)

Synchronous Dataflow (SDF) [48] models of computation have often been used by developers to model Digital Signal Processing (DSP) systems, as they offer a natural abstraction for block-diagram languages. An SDF model consists of interconnected nodes with arcs, where each node represents an *actor* and the arcs represent the data paths. Data tokens, the units of data in an SDF model, flow along the data paths in and out of actors. Actors implement the computations dictated by the system being modeled. When input data is available for an actor, it gets enabled and is fired. The actor then executes consuming a finite number of tokens and producing a finite number of tokens. In an SDF model of computation, the computation and the communication is scheduled statically. This implies that, the when a system is modeled as an SDF graph, it can be converted into an implementation that is guaranteed to complete all the tasks in finite-time and using finite memory. Also, once the required amount of resources are made available, SDF graphs can be executed over and over again in a periodic fashion without any additional resources. This makes SDF an ideal candidate to model DSP systems. Figure 10 shows an example SDF graph model with the actors annotated with number of input and output tokens they consume and produce. Ex: *Actor 1* consumes 1 token but produces 2 tokens.

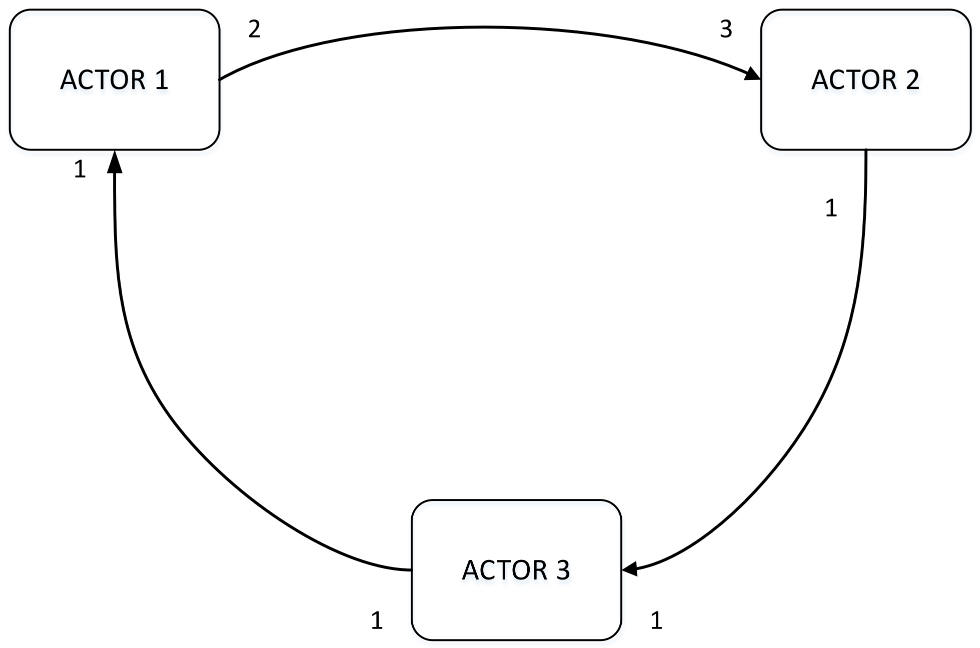


Figure 10: Example of Synchronous Dataflow Model

A system modeled using SDF graph uses FIFO for communication purposes and they are ideally suited for expressing algorithms with one-dimensional (1-D) data. For describing algorithms which involved multi-dimensional data, first the data was made one-dimensional and then SDF graphs were used to describe the algorithms. To avoid this, multi-dimensional SDF [58] was proposed. MD-SDFs used arrays in place of FIFOs and this eased the work of describing algorithms that involve multi-dimensional data. The scheduling algorithms of SDF were further generalized for MD-SDFs.

2.3.3 Petri Nets

Petri Net [63, 57] is a specification language proposed by C.A.Petri and is used for describing and studying distributed systems. A Petri Net consists of nodes and arcs. Nodes are of two types - *Places* and *Transitions*. In a Petri Net model, places represent the conditions and transitions represent events. Arcs are present from a Place to a Transition or from a Transition to a Place. A Transition occurs when the pre-conditions and the post-conditions associated with its input and output are satisfied. Unless an execution policy is defined, the execution of Petri Net is non-deterministic. Petri Nets allow efficient discovering of concurrency and synchronization points as compared to C/C++ programs. This will allow better task partitioning and task scheduling in distributed systems. Graphically a Petri Net is represented as a directed graph with a node marked as initial state. An example Petri Net representing water composition is shown in Figure 11.

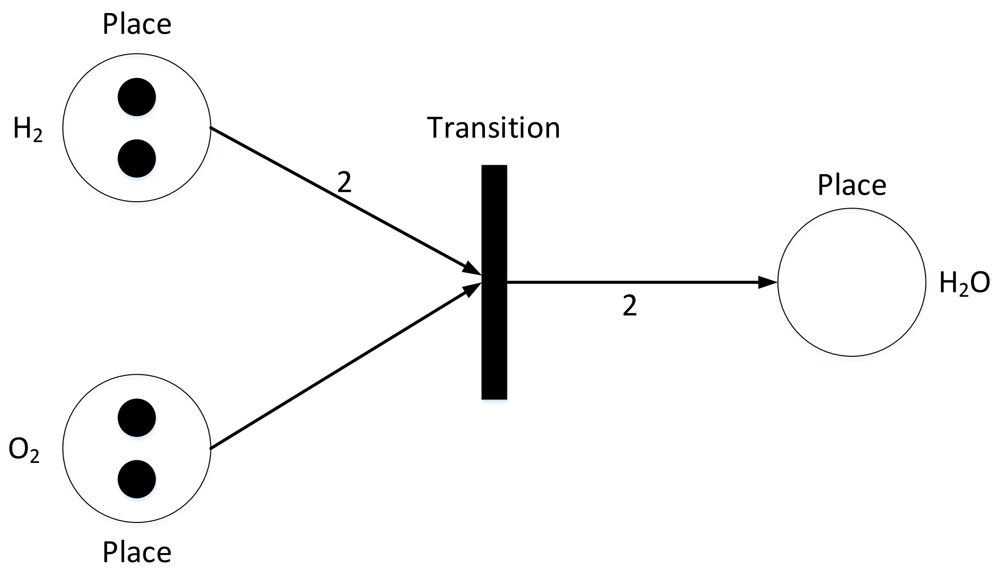


Figure 11: Example of Petri net representing water composition

One of the major problem that arises with the use of Petri Nets is *state explosion*. Since the Petri Net is very generic and simple, even a small system would require many states and transitions, which will result in a huge number of states for a complex system and might render the Petri Net unsolvable. Another problem with Petri Nets is the abstraction level at which the nets describe the intended system. If its too close to the implementation model, then discovering concurrency might be difficult. Petri Nets, as with any graphical formal specification language, also faces the issue of managing large scale models.

2.3.4 Input/Output Automata

Input/Output (I/O) Automata [30] is a formal model which is often used to describe behavior of asynchronous concurrent systems. Having formal semantics, models described as I/O automata could be subjected to formal analysis and one can reason about the system that is modeled.

2.3.5 Ptolemy Framework

Ptolemy framework [24, 28], developed at University of California Berkeley, is an open-source framework that is used to model, simulate and design concurrent real-time heterogeneous embedded systems. Ptolemy supports actor-oriented design. A Ptolemy model is a hierarchical interconnection of actors. Actors are basically components that execute concurrently and communicate using messages sent and received via ports. The semantics of the ptolemy model is not determined by the Ptolemy framework, but it is determined by the *director* component of the model. The *director* is a software component which implements a model of computation. The directors in Ptolemy support various model of computations such as KPN, Discrete-Event (DE), Continuous Time (CT), Synchronous Dataflow (SDF), etc. Each hierarchical level in the Ptolemy model can have its own director, and each of these directors can implement a different model of computation, and they can all be composed together.

These features make Ptolemy a very useful framework to model and simulate heterogeneous embedded systems.

In Table 6 we list some of the popular commercial tools that used in embedded system development.

Table 6: Popular Commercial Tools used for Modeling Embedded Systems and Code Generation

No.	Tool	Graphical/ Textual	Semantics	Highlights
1	LabView[79]	Graphical	Non-formal	Predominantly used for generating code that controls instruments
2	Simulink[83]	Graphical	Non-formal	Modeling, Simulating and Analyzing multi-domain dynamic systems
3	Stateflow[84]	Both	Semi-formal	Used to model control systems within Simulink models
4	Modelica[80]	Textual	Non-formal	Declarative, component-oriented, modeling language
5	xcos[82]	Graphical	Non-formal	Used to model and simualate the dynamics of hybrid systems

3 Results and Discussion

3.1 Related Work

In Section 3.1, we discuss some of the works which are closely related with our work. This Section 3.1 is organized similar to the thesis organization. First, we explain the works related to trusted software synthesis, specifically, sequential and concurrent software synthesis works. Second, we discuss the works related to formal ASIP synthesis. Finally, we explain the works related to causality analysis, type checking and value range analysis in modeling languages.

3.1.1 Concurrent Software Synthesis

Numerous efforts have been made in the past to synthesize code from synchronous specifications. But most of these efforts were targeted towards generating sequential code rather than multi-threaded code. Here, we list some of the multi-threaded code generation efforts. The authors of [78] proposed an approach to generate multi-threaded code from Esterel specifications. Their approach involved partitioning of concurrently executable Esterel statements into communicating Finete State Machines (FSMs) and distributing the computation of these FSMs based on the communication and synchronization techniques used in reactive processors. In [7], the authors provide a way to translate synchronous guarded actions to multi-threaded C code. They build an action dependency graph using the synchronous guarded actions, extract concurrently runnable tasks from the graph and map them to threads. Both these works are targeted at single clock systems while our work focuses on systems with multiple clocks (polychronous). In [41], the authors provide a non-invasive methodology which includes generating programming glue to generate multi-threaded code from polychronous specifications. This approach requires that, no variables are shared between the concurrently executable processes, in other words, the clock trees of sub-processes do not intersect. This a big limitation and generating multi-threaded code for independent processes is very trivial. In another similar work [64], the authors focus on generating multi-threaded code for mutually independent tasks, which is trivial. In [76], the authors have explained the concept of weak-hierarchy and composition of endochronous processes. Using these concepts one can identify parts which can be concurrently executed without disabling one another. This work also lists some of the rules for composing endochronous systems to a weakly endochronous system. To the best of our knowledge, there is no implementation of this. The technique explained in Section 3.3 considers and extends the theory presented in [76]. We propose a novel efficient technique by which we can test concurrent implementability of a given MRICDF model by decomposing it. Our technique also generates execution schedule and the multi-threaded code that conforms to the schedule. We present all the algorithms involved and investigate the feasibility and scalability of the proposed technique.

The authors of [73] first proposed the idea of affine transformations for Polychronous language SIGNAL as an extension to allow specification and validation of real-time systems. They further explored the domain of affine relations in the follow up work [72]. We have used the concept of affine clocks to identify and remove avoidable synchronizations and improve the efficiency of the synthesized multi-threaded code.

3.1.2 ASIP Synthesis

In the past, efforts such as [11], [71], [54] etc., have been made to generate hardware from Simulink models, synchronous languages such as Esterel, Lustre, SIGNAL and others. However, most of these efforts have concentrated on design exploration, verification of certain properties of the system being designed and synthesizing register-transfer level VHDL/Verilog logic. Synthesizing ASIPs with maximal concurrency and resource sharing among multiple complex instructions is a more recent interest, as this allows one to co-design application specific processors to speed up computations that are off-loaded from main processors. The work presented in this paper closely resembles the works in [46] and [47]. In [46], the authors have proposed a methodology based on preliminary experiments to generate efficient HDL code from SIGNAL specifications. Their approach consists of applying multiple semantic preserving transformations to original SIGNAL specifications. Efficiency of HDL code stems from the fact that they use the SIGNAL clock exclusivity information to identify re-usable resources during one of transformations. The authors of [47] have used *Hierarchical Conditional Dependency Graph*(HCDG) as intermediate representation and proposed a unifying approach to determine mutual exclusiveness and scheduling conditional behaviors. In [47], the authors also list other works that have addressed the problem of efficiently scheduling conditional behaviors. Our work differs from the above mentioned works in the sense that, we provide a new compilation scheme for SIGNAL/MRICDF programming languages based on conditional partial order graphs, which is a natural fit for control state optimization, and scheduling of control states in the ASIP. The sequencing of control states is akin to micro-programming, but it is automatically synthesized.

3.1.3 Verification and Validation

3.1.3.1 Causality Analysis

Polyhedral domain as an abstract domain for abstract interpretation has been used by many since the seminal paper on abstract interpretation by Cousot and Halbwachs [19]. In [6], a survey on various applications of Polyhedral analysis in program verification can be found. In the context of Polychronous model of computation, the only work we are aware of is [13], where the authors showed a polyhedral abstraction domain for Signal programs to prove formal properties. However, to the best of our knowledge, no implementation of Signal verification based on this work exists. Detection of causal cycles in programs has been extensively researched. In one of the early Polychrony related papers [51], the authors explored the requirements for a Signal specification not to exhibit causal behavior. The important one is checking that in any cyclical dependency in the dependency graph, the conditions that make each dependency active, must not all evaluate to true during the same logical instant. Later in [8] and [10], algorithms for checking whether the apparent cycles are causal have been discussed. However, all these past work have a limitation that they work only on Boolean abstraction of the predicates (conditions in conditional dependencies), and hence sound but imprecise. Even such sound but imprecise detection problem is NP-complete, and hence only heuristics can be used. However, making this analysis both sound and exact, is easily proven to be an undecidable problem. To the best of our knowledge, there is no synthesis tool for polychronous specifications that can provide the safe operating range for inputs where causal behavior is not exhibited by the system. In [40], the authors perform causality analysis of MRICDF(Multi-rate Instantaneous Channel connected Data Flow) [43] specifications by using an Yices SMT solver as a constraint solver. In another work [39], the authors identify reachable and non-reachable causal loops by converting the MRICDF specifications to Quartz specifications and in turn generating SMV files for model checkers. This technique does not consider input constraints, is expensive and suffers from state explosion problem. In this paper, we go further than any of the above works. If true causal loops exist, our technique also provides the range in terms of inputs where the causal behavior is not exhibited, i.e., the bounds in terms of inputs for safe operating region, albeit, it does not admit all possible input from the original input space. If the user is given a choice to either reject the entire specification or to generate code that operates safety, and the user chooses the latter, our compiler provides the solution. Finally we also propose a wrapper extended code synthesis for the system so as the guarantee that the synthesized code always executes in safe operating region.

3.1.3.2 Type Checking

There has been a substantial amount of research done in the area of type checking in software. But most of these works correspond to the analysis done either statically on the written software or dynamically by the software itself. Recently, there has been a shift ([26]) in focus towards model-based engineering to address type inconsistencies in critical software. In this work, we propose to do the analysis on the formal MRICDF ([43],[42]) models rather than the actual software. The authors of [45] explain about the polymorphic (union) dimension types and their parametric nature. A related work is [2], where the authors type annotate certain signals of the charon specifications with dimension & unit information and further try to infer the same for other signals. Another closely related work is [50], in which the authors propose a framework to create ontologies representing user-defined domains and check for dimensional and unit consistencies. In another related work [67], the authors extend the type system of Simulink modeling language with dimensional and unit information associated with ports and signals. They capture invariants w.r.t the types and transform these invariants into formulas. These formulas are used to check well-formedness of the Simulink blocks using yices ([86]) SMT solver. In all these works, the authors assume that signals are not of union (polymorphic) type. This could be a major limitation in most of the modern designs as multiplexing is very commonly used. Verifying correctness becomes difficult when union typed signals are involved. In [74], the authors explain the clocks in synchronous languages and how they can be used for checking type soundness. The authors of [22] propose a type assignment scheme for union types in call-by-value languages. In [15], the authors present a method to infer type qualifiers by generating constraints in presence of user-defined rules. Even though the central idea of our work resembles the ideas proposed in [2], [67] and [50], our work is very different from them. We have addressed the shortcomings of the previous works by extending the type system of MRICDF language to support both non-union and union types. We absorb ideas from [74], [22], [15] etc. and propose inference rules for inferring the type information for MRICDF models. An inference algorithm is provided to infer the type information for all signals. In case of union types, we use the clock of signals to infer which individual type occur under what constraints.

We then use these constraints and derive other invariants from the MRICDF models and check for consistencies using yices - an SMT solver. Our approach is sound and addresses shortcomings of the previous approaches.

3.1.3.3 Value Range Analysis

Value range analysis techniques, especially, integer value range analysis techniques has been well explored in the past for synchronous programming languages such as C, C++, $Java$, etc. The fundamentals for range analysis were first proposed by Cousot and Cousot in their seminal paper [17]. This was further expanded by them in [18]. Based on their work, various researchers proposed numerous other range analysis algorithms. Clark et al in [16], and Flanagan et al in [27], developed theories that uses the fundamentals proposed in [17] to statically identify the errors in programs. The authors of [70] proposed a way to detect buffer overflow vulnerabilities in C programs by doing range analysis. In [66], the authors proposed a range analysis technique for C programs that provides a balance between speed and accuracy. The commercial Polyspace tool [36] allows data range specification and further, the tool finds bugs related to out-of-array access, integer overflow, etc by conducting abstract range analysis. All of these research efforts are targeted towards synchronous programming languages and not for polychronous programming languages. The polychronous programming model introduces additional complexities which would require extensions to the traditional analysis techniques. This makes our proposed approach novel and thus, it completely differs from all of the previous works.

There have also been attempts at type system extensions and abstraction-based analysis to tackle similar problems. In [50], the authors propose a static analysis technique that uses lattice-based ontologies to analyze Ptolemy models. In a closely related work [60], the authors have proposed a similar polychronous type extensions to do specify units and dimensional information and proposed techniques to verify the dimensional consistencies. In another closely related work [29], the authors propose a Boolean abstraction to polychronous models. They further use the abstraction along with clocks to do determine reaction absence and generate efficient code. The authors of [59], used abstractions based on polyhedra to abstract polychronous models and statically determine if a given causal loop is constructive or not. We inherit the ideas from [29], [36], [60], [59], and extend them in this work. Our proposed technique includes clock calculus as part of inference algorithm, which further improves the accuracy of analysis. To the best of our knowledge, this is the first attempt at doing value range analysis of polychoronous models. This combined with the SMT-based verification renders the proposed approach to be extremely useful in proving properties related to signal value ranges of signals in polychronous models.

3.2 MRICDF Polychronous Formalism and EmCodeSyn Synthesis Tool

3.2.1 Definitions and Overview of Concepts

A Multi-Rate Instantaneous Channel-connected Data Flow (MRICDF) model is a data flow network model that consists of several synchronous modules called *actors*, that are interconnected using *channels*. An actor represents a computation with an input interface and an output interface for input and output signals respectively. Actors communicate with each other via channels using *signals*. Communication is instantaneous and channels can have different communicating rates. In all, a MRICDF model represents a network of synchronous modules with multiple clocks, which is the basic definition of a polychronous system.

In the polychronous model of computation, *events* form the primitive entities. An *event* occurs whenever there is a change in the value at an input or output port, or change in value of a variable etc.

Definition 1 ***[Event]*** *Let Ξ, to denote the set of all events, and $\leq$, denote a preorder relation among events which indicates the precedence of one event over another. $\leq$ is a preorder on Ξ: $e \leq f$ means that, event e occurs before or concurrently with event f. $:$ is the equivalence relation based on $\leq$: $e : f$ means that, events e and f occur simultaneously, also called as synchronous events.*

A logical instant is a maximal set of computations that occur in reaction to one or more events. This set of computations is maximal in the sense that, any other activity would require another value to arrive on those inputs which triggered the current set of computations. Events within one logical instance are all synchronous with each other.

Definition 2 ***[Logical Instant or Instant]*** *Let Υ denote the quotient of $\Xi / :$, the set of logical instants. Thus a logical instant is a maximal set of events that are synchronous.*

The synchronous events within a logical instant may be constrained by data dependencies and hence are also partially ordered by a data-dependence relation. All the dependency relations are captured in the data dependency graph. An implementation of the data-flow specification is only possible if the dependency relations do not form a cycle, since it is used to schedule the order of computation within each logical instant during code generation phase.

Definition 3 ***[Data Dependency]*** *We use $\rightharpoonup$ to express data dependency between events. The binary relation $e \rightharpoonup f$ means, e has to be computed after f, in other words, f precedes e.*

If the relation holds between some pairs of synchronous events of two signals, then the data dependency relation is elevated between those signals. Note that a subset of events of one signal may be data dependent on a subset of events in another signal. If there is a Boolean condition c, such that these data dependencies holds iff c is *true*, then we say that the data dependence between the two signals hold under c. $\forall$ signals x, y and c, ($\forall t \in \Upsilon$, $c(t) \leftrightarrow (x(t) \square y(t))) \Rightarrow x \rightarrow^{c} y$.

Definition 4 ***[Signal]*** *A signal is a stream of values that occur at specific instants.*

Let T be the *type* representing set of values a signal can take, $\perp$ be a special value used to denote the absence of the signal, and $T_{\perp} = T \cup \{\perp\}$, then we can define a *signal* as a function $\Upsilon \rightarrow T_{\perp}$.

For a given signal x, there exists one maximal set of logical instants $\gamma \subset \Upsilon$, such that γ is a total order in Υ and the signal x takes a value from T in each of the instants of γ. Such a set is called the epoch of the signal represented by $\sigma(x)$. The term *epoch* and the term *clock* are interchangeable in the same way a set and its characteristic functions are interchangeably used depending on the context.

Definition 5 ***[Clock]*** *The clock of a signal is a characteristic function that tells if a signal x is present or absent at any given instant t in Υ.*

Clock is a function $(\Upsilon \rightarrow T_{\perp}) \rightarrow \Upsilon \rightarrow \{true, \perp\}$ that for a signal x returns another signal $(\hat{x})$ defined by: $\hat{x}(t) = \mathrm{true} \, \mathrm{if} x(t) \in T$ and $\hat{x}(t) = \perp \mathrm{if} x(t) = \perp$.

Definition 6 ***[Epoch]*** *The epoch of a signal is a set of all logical instants at which the signal is computed or assigned new values.*

While the set of logical instants Υ is partially ordered, the epoch of a signal is a totally ordered set. The clock of a signal is a boolean signal that takes the value *true* at every logical instant where the signal has a value, and is absent in all other instants. Not all signals at the interface of a process are present and computed or assigned input values at every logical instant. Thus signals may have different clocks – hence the model of computation is called polychronous or "multi-clocked".

Definition 7 ***[Clock tree]*** *Using the clock relations, a hierarchy of clocks can be built and the resulting hierarchical structure is a clock tree or a forest of clock trees depending on whether the hierarchical structure is single rooted or multi-rooted.*

Based on the above definition, signals can be classified into,
• signals x and y are synchronous to each other if their clocks are same: $\hat{x} = \hat{y}$.
• if signal x has events in a subset of instants where signal y has events, then $\hat{x}$ is a sub-clock of $\hat{y}$.
• if signal x and y do not have events that belong to same logical instant, then their clocks can be either mutually exclusive or they are unrelated.

The information regarding clocks of all signals is stored in *clock tree*. If a set of clocks { $\hat{x}_1 \cdot \cdot \hat{x}_n$ } are related by a Boolean relation R, then in order to relate how an arbitrary logical instant $T \in \Upsilon$ may be shared between them, we can relate Boolean variables $b_{x_1},...,b_{x_n}$, where $b_{x_i} \leftrightarrow T \in \hat{x}_i$ by a Boolean relation R' where R and R' are isomorphic. For example, if $\hat{x} = \hat{y} \cup \hat{z}$, then we can write $b_x = b_y \vee b_z$. Thus from a set of clock relations, we obtain a set of Boolean relations.

3.2.2 MRICDF Actors

Actors in MRICDF language can be classified into two groups – (a) Primitive actors (b) Composite Actors. The four primitive actors are,

• **Function Actor:** This actor performs any user specified computation in any instant when the inputs have an event. All the inputs and outputs are synchronized with each other.

Operation: $r = a * b$

Clock relation: $\hat{r} = \hat{a} = \hat{b}$

Boolean relation: $b_r = b_a = b_b$

Dependency relation: $r \rightarrow a$, $r \rightarrow b$

• **Buffer Actor:** This actor is used to temporarily store a value of a signal across instants, in other words – it delays a signal. The signal must have events in both storing and retrieving instants. Increasing the buffer size of the Buffer actor produces the same effect as a series of unit sized Buffer actors cascaded. Both input and output are synchronized with each other.

Operation: $r = b$ \$ $n\ init\ v_1..v_n$

Boolean relation: $b_r = b_b$

Clock relation: $\hat{r} = \hat{b}$

• **Sampler Actor:** This actor is used to down-sample a signal based on a known Boolean condition. This actor produces outputs in all instants where there is an input and the Boolean condition evaluates to $true$. Hence the output clock is the intersection of input clock and the clock when Boolean condition is $true$.

Operation: $r = a\ when\ b$

Clock relation: $\hat{r} = \hat{a} * [b]$, where $[b] = b$ is $true$

Boolean relation: $b_r = b_a\ and\ b_{[b]}$, $b_b = b_{[b]}\ or\ b_{[\bar{b}]}$, $b_{[b]}\ and\ b_{[\bar{b}]} = false$

Dependency relation: $r \rightarrow a$

• **Merge Actor:** This actor merges two signals (can have different clocks) with a higher priority for one of the signal. The clock of the output signal is the union of the clocks of the participating input signals.

Operation: $r = a\ default\ b$

Clock relation: $\hat{r} = \hat{a} + \hat{b}$

Boolean relation: $b_r = b_a \ or\ b_b$

Dependency relation: $r \to a, \quad r \to b$

Composite actors are hierarchical combination of several primitive actors.

3.2.3 Master Trigger and Sequential Implementability

Given an MRICDF model, we need to translate it to an executable implementation which is *latency equivalent* to the MRICDF model. However, before translation one has to ensure if the given model is implementable as deterministic and equivalent software or not. It might be sequentially implementable, or concurrently implementable, or not implementable as deterministic equivalent software. To do so, we have to identify the mapping from the abstracted MRICDF entities to actual software. Out of the many mappings, we discuss two important ones here. First one is the mapping of the logical instants consisting of synchronous but data-dependent events to a set of sequential computation steps. If the logical instants are totally ordered, then each logical instant can be mapped to an iteration of a loop. If logical instants are partially ordered, one has to check if multiple sequential ordered chains can be mapped to multiple threads – but in that case, the synchronization on shared events must be deterministically achievable. If the logical instants are totally ordered, then there exists one or more signals that are present in event logical instant of the model. If there are multiple of these, they must be synchronous to each other. Therefore, their data dependency partial order must be analyzed to find the signal which forms the least element in the partial order. Such signal is called as –*Master Trigger* [43]. To identify a signal which can be *master trigger*, we first construct the Boolean formulae from the clock relations as explained before. In [43], we show that computing the *master trigger* for the MRICDF model is equivalent to identifying *unitary positive prime implicate* in the constructed Boolean formula. We use a SMT-based technique to identify the prime implicates. If the model is not sequentially implementable, a unitary positive prime implicate does not exist. An iterative computation of prime implicates allow us to construct subclock relationships between clocks of all the signals. If this subclock relationship forms a tree, then we go on to the scheduling phase.

The scheduling phase maps the events within each logical instant to computation within an iteration. The order of computations is constrained by the data dependencies implied from the specifications. If the dependency relation is not a partial order, and has a cycle, we cannot find a schedule. This analysis is also done using SMT-based techniques as explained in [40],[59]. In the Section 3.3, we show a novel method to find out if an MRICDF model is concurrently implementable or not.

3.3 Synthesis of Multi-Threaded Code from Polychronous models

Consider an automotive cruise controller system implemented based on a proportional integral (PI) control, with vr as the target cruise speed, v as the actual sampled speed, T as the number of samples between the computation of two subsequent thrust (actuation) outputs u and e as the difference between targeted speed and the actual speed. The pseudo-C code for this system is shown in the Listing 3.3.1, where S is the local variable accumulating the integral and ki are k are constants determined based on PI control. In this pseudo-C code, Sample(v), and Output(u) are input/output actions. Now, consider the control loop for the temperature (AC) control system in the same car. Assuming PI control paradigm, the pseudo-C code for temperature control is shown in the Listing 3.3.2, where S is integration summand and ci and c are constants.

Listing 3.3.1 Cruise Control System

```
L : S = 0;
Thrust_Interval = T;
while(Thrust_Interval != 0){
        Sample v;
        e = vr - v;
        S = S + e * ki;
        Thrust_Interval = Thurst_Interval - 1;
}

Sample v;
u = k * (vr - v ) + S;
Output(u);
GOTO L;
```

Listing 3.3.2. AC Control System

```
L : S = 0;
AC_Interval = T’;
while(AC_Interval != 0){
        Sample p;
        e = pr - p;
        S = S + e * ci;
        AC_Interval = AC_Interval - 1;
}
Sample p;
w = c * (pr - p ) + S;
Output(w);
GOTO L;
```

In the AC controller system, p denotes the currently sampled temperature, pr denotes the target temperature set by the thermostat, w is the signal whose value controls the valve aperture to release hot air or cool air, and speed of air. Note that the AC control loop and the cruise control loop might be working at different sampling rates, and their actuation intervals (Thrust_Interval and AC_Interval) could also be different. If both these control loops are run on the same processor, and scheduled using a real-time scheduling algorithm (with T and T’ being the respective deadlines, and periods for the two tasks), one could easily implement them as two real-time processes. As these two processes do not have any interaction, there is no dependency or no need for any synchronization, and in that case the job of the embedded software designer is simple. Now, consider the hypothetical possibility that – whenever the sampled temperature goes below a certain threshold, the cruise control is to be disengaged to manual control, because such low temperature might be indicative of icy weather conditions. This is not necessarily an ideal automotive design example, but rather concocted to make a point regarding multi-threaded control. If the temperature loop is tasked to generate an interrupt and the interrupt is input to the cruise control loop to disengage it, then we have two processes or threads which interact, and timely response to the interrupt needs to be guaranteed. The pseudo-C code for both control systems with interrupts is shown in Listing 3.3.3 and 3.3.4. In this code, we assume that the intrpt is the name of a single bit buffer, whose value is set atomically to *true* or *false*, depending on if the temperature control wants to send interrupt or not. Since this is shared buffer, a semaphore mechanism is assumed to synchronize the read/write of this buffer. The semaphore effectively enforces a barrier synchronization between the two control threads at their outer loops.

Listing 3.3.3 Cruise Control System with Interrupts

```
L : S = 0;
Thrust_Interval = T;
while(Thrust_Interval != 0){
        Sample v;
        e = vr - v;
        S = S + e * ki;
        Thrust_Interval = Thurst_Interval - 1;
}

Sample v;
Sample interrupt;
if (interrupt = true)
interrupted = true;
else interrupted = false;
if (interrupted == true)
u = k * (vr - v ) + S;
Output(u);
GOTO L;
```

Listing 3.3.4 AC Control System with Interrupts

```
L : S = 0;
AC_Interval = T';
while(AC_Interval != 0){
        Sample p;
        e = pr - p;
        S = S + e * ci;
        AC_Interval = AC_Interval - 1;
}
Sample p;
w = c * (pr - p ) + S;
interrupt =    (p < P)? true: false;
Output(w);
GOTO L;
```

This is too simple an example, and hence, getting this synchronization correct is trivial with the use of a single Boolean semaphore. However, multiple threads may need to synchronize at various places of their execution with different threads. In order to be predictable, and safe, the system's behavior must be deterministic. Guaranteeing determinism with multiple synchronizations among a group of threads, while also ensuring no deadlock, is often hard and error prone. Anyone programming multi-threaded code of reasonable complexity would have faced many bugs, and debugging cycles.

One way to guarantee determinism is to tightly synchronize all the threads with barrier synchronization after every step. However, that is not recommended for obvious lack of efficiency of such code. For reasonable performance, threads responsible for distinct control functions must make progress asynchronous to each other except when they *must* interact. In the above example, the synchronization is done at the outer loop of the control and not at the sampling loop. Synchronization at every sampling loop iteration would have made the sampling rates in the two threads dependent on each other and slow down progress. Also, depending on the nature of the physical signal being sampled, they may require different sampling rates. We also want to make it easy for designers to decide which variables are to be shared (in this case the intrpt), and ensure that when a new value is produced, it is eventually read by the other thread, and that it does not read the same value twice. We also do not want that the absence of an interrupt hold up the other thread too long, and hence absence is encoded as $false$. Such decisions can be taken by the programmer while programming in C or other programming languages, but then proving correctness (i.e. to prove that synchronization indeed guarantees that every interrupt is responded to, and absence of interrupt does not hamper progress, and that there is no deadlock) is much more involved – especially when the number of threads and number of synchronization points are numerous. If we can capture these requirements in a simple formal model, write appropriate constraints, and generate multi-threaded C-code with appropriate synchronization code, and the code-generation is provably correct, the validation overhead can be shifted to validating the formal model which often is much easier and computationally less expensive.

In synchronous programming languages such as Esterel [12], Quartz [68], or Lustre [34], these two loops will be modeled as two distinct processes since they do not need to move forward by synchronizing at every macrostep. If the two threads were modeled in standard Esterel, it has to be over designed by making two parallel synchronous threads that synchronize quite tightly. To achieve the independence of the sampling rates and the thrust generation intervals by the two loops, one has to model them as separate independent processes. Therefore, the proof of determinacy of interaction for these have to be reasoned at a meta-level. The interaction between the two processes will be external to the model of the processes. Thus proving the correctness will also be done outside the code-synthesis step. Since our goal is 'correct-by-construction' code synthesis, ideally, the code-synthesis step should guarantee 'correctness' without external reasoning about the generated code. Therefore, we have chosen polychronous modeling paradigms such as signal or MRICDF, and developed techniques for multi-threaded code synthesis. Our synthesis approach produces correct synchronization between two asynchronously progressing threads, with synchronization on a need basis, and guarantees determinacy.

In order to model these two control loops in MRICDF or signal, we first create two subprocesses, with the cruise control subprocess having an extra Boolean interrupt as a shared variable intrpt. When there is a temperature constraint violation, the second subprocess will set this shared interrupt variable to $true$, otherwise it will set the value to $false$. All we have to specify is that the two subprocesses synchronize on reading and writing of this variable, and rest is taken care of during the code generation. The code generation first needs to prove that the synchronization can be done deterministically, and without possibility of deadlock. Only then, it will progress to generate code, by adding synchronization primitives.

This entire process is done by a simple clock calculus on these processes. In the first process, the clocks of v, S, e are the same, whereas the clock of intrpt is a subclock, which is the same clock as that of the thrust output u. In the second process, p, S, e have the same clock, which is possibly distinct from the main clock of the other process because the temperature sampling may be less frequently done than speed sampling. However, an additional constraint in the MRICDF or signal model will be provided in the model which says that the clock of intrpt must synchronize. This statement states that the AC control process must rendezvous with the cruise-control process when it is time to read/write the interrupt. Thus, the two processes can be implemented with two separate threads, which will only synchronize on an interrupt. This multi-threaded process will be deterministic – that is, for the same flow of input events on the sampled speed and temperatures, same flow of outputs will occur. One could make other design decisions such as not reading or writing interrupt that often, so they could consider creating further conditions for read/write of the interrupts. However, for the correct synchronization synthesis to work, both the processes must independently be able to compute such condition. For example, if they are supposed to exchange interrupt information every n times thrust is generated, and every m times temperature control actuation is generated, that is easy to express as well.

Novelty in Our Approach

As we have argued, the polychronous (multi-clock) nature of signal/MRICDF, will allow the system to be modeled as a single process and yet yield to 'correct-by-construction' multi-threaded code generation. Also, the reasoning about determinism can be done on the whole system, without the need for making any meta-level assumptions on the occurrence of interrupts, as the reasoning will be embedded in the polychronous clock calculus.

A graphical tool, EmCodeSyn[42] analyzes the MRICDF models and checks for implementability before generating code. EmCodeSyn currently can only check for sequential implementability by conducting, a static *Epoch Analysis*/*Clock Calculus* (explained later) on a set of Boolean equations derived from the MRICDF model. This analysis is based on the Boolean theory and prime implicates [43]. In Section 3.3, we address – how to automatically generate deterministic multi-threaded code that is "correct-by-construction". We extend the capabilities of EmCodeSyn tool, with a novel technique for checking the concurrent implementability of MRICDF models. We particularly focus on efficiency of the generated code and the practicality of the proposed approach. The proposed technique involves identification of systems that are *weakly-endochronous* [62]. If found implementable, the technique further generates the execution schedule and multi-threaded code with appropriate synchronization constraints that conforms to the schedule. We have implemented these in EmCodeSyn tool and conducted experiments to test performance and scalability issues. It should be noted that a similar idea could be used for generating multi-threaded code for systems specified using signal language as well. In fact, the theory of weakly hierarchical processes developed in [76] for signal forms the basis of our work. However, other approaches to multi-threaded code generation from signal are quite different. The approach in [62] approaches the problem with extreme fine granularity by enumerating all possible reactions and computing dependence, and the resulting complexity of the synthesis is very high. We focus on identifying threads that are rooted at distinct incomparable clocks in the clock hierarchy. The difference between these two approaches can be summarized as *fine-grained concurrency* vs. *task level concurrency*.

Contributions detailed in Section 3.3:

1. A novel technique for determining concurrent implementability of the MRICDF models based on prime implicate theory
2. Technique for generating execution schedule and multi-threaded code with

appropriate synchronization constraints for implementable models
3. Experimental results showing the scalability of the proposed technique and comparing efficiency of the generated code as compared to hand written code

Consider a simple MRICDF model shown in Figure 12(a). For readability purposes, its textual representation is shown in Listing 3.3.5. It has 2 input signals v and u and 1 output signal w. Internal signal x's value is computed in instants where v has events. Similarly y is computed in every instant where u has events, but w is computed in instants only when $x >= 10$, $y >= 20$ and both u and v have events. Hence we can say that, $x : v$ and $y : u$ as $\hat{x} = \hat{v}$ and $\hat{y} = \hat{u}$. The subclock relationship for this model is shown in Figure 12(b), where each node represents a unique clock. The labels on the arrows indicate the constraints on the values of signals that gives rise to the subclocks. Nodes with multiple incoming edges, represent a clock which is a subclock of multiple distinct clocks, and it actually contains only those instants of the parent clock where the constraint on the incoming edges are satisfied. It must be noted that this clock tree is not single rooted. This means that there is no single signal which could be used as a *master trigger*. If our aim was to generate sequential code, then we can synthesize a temporary signal that has events when either u or v or both have events, and then use this temporary signal as *master trigger*. With the addition of this temporary signal to the clock tree, it becomes single rooted. But such modification is a refinement of the original model, and only represents a subset of the behaviors of the original model. But here, our aim is to synthesize multi-threaded code which contains all the behaviors of the original model. For multi-threaded code, instead of a single *master trigger*, we have a set of *partial triggers*. Each of these *partial triggers* act as *master trigger* for an individual thread. There can be logical instants where some of the partial triggers are present and some are absent but there cannot be any logical instant which all the partial triggers are absent. Clock tree for such a system will have multiple roots as shown in Figure 12(b).

Definition 8 ***[Partial Triggers]*** *Let P be a MRICDF model representing a data flow process and let y be any signal in the process. A set of signals $S = \{x_1, .., x_n\}$ is the set of partial triggers for P if it is maximal such set with the following two properties :*

- $\forall y \notin S, \exists x_i \in S$, such that y is present x_i is present, i.e, $\sigma(y) \subseteq \sigma(x_i)$ for some $x_i \in S$

- $\forall x_i, x_j \in S$, $(x_i \neq x_j)((\sigma(x_i)\text{Ú}\sigma(x_j)) \wedge (\sigma(x_j)\text{Ú}\sigma(x_i)))$

Listing 3.3.5 Textual Representation for MRICDF model in Figure 12

```
process P = (? integer u,v; ! integer w)
( |x := x$ init 0 + v
| y := y$ init 0 + u
  | w ^= (x >= 10) ^= (y >= 20)
  | w := x + y when w
  |)
```

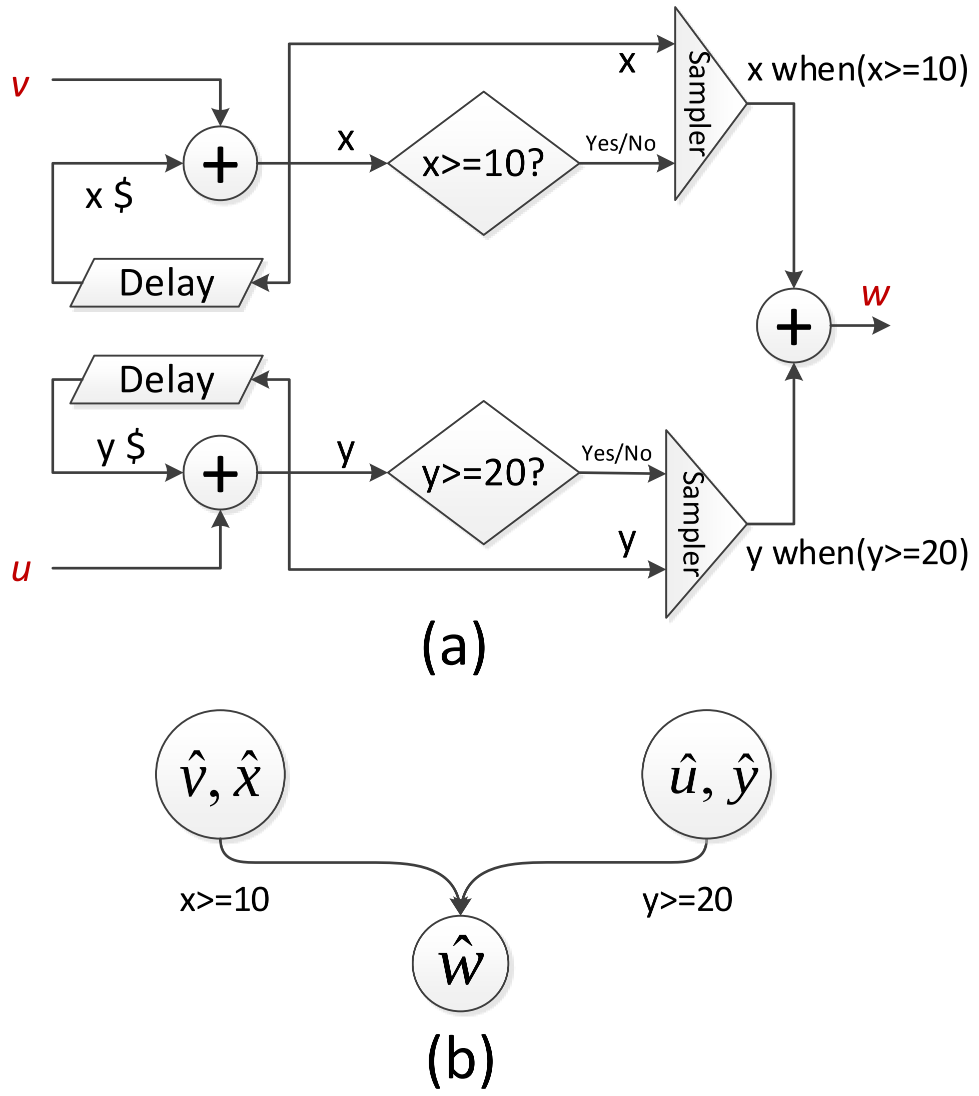


Figure 12: (a) MRICDF model, (b) simplified clock tree

3.3.1 Constraints for Concurrent Implementability

In [76], the authors define a class of processes for which concurrent deterministic implementation is guaranteed. This class consists of processes composed of individual sub-processes with their own triggers. A list of conditions that identify those processes was identified. Let P, be a MRICDF model representing a data-flow process that consists of numerous sub-processes. P can be scheduled concurrently if,

1. The process P can be partitioned into multiple sub-processes $\{P_1,..P_n\}$ and $\{x_1,..x_n\}$ represent their respective master triggers.
2. The dependency graph of the process P does not have cycles.
3. P is well-clocked: the relations between epochs inside subprocesses are compatible at the level of the process. In other words, scheduling of sub-processes does not result in a deadlock.

Considering those rules, we define the criteria for concurrent implementability as follows–

• For each signal $y \in P$, there exists at least one partial trigger $x \in \{x_1,..x_n\}$, such that epoch of y is a subset of and can be derived from epoch of partial trigger x, i.e, $\forall y \in P, \exists x$ such that, $\sigma(x) \supseteq \sigma(y)$ and $\exists f$ – a Boolean function such that for each $t \in \sigma(x)$, whether $\hat{y}(t)$ is true can be determined by computing f on values of the signal x and other signals that are read before y in instant t.

• Non-existence of any cyclic causal loops for any instant can be proved by constructing SMT instances from the edges of clocked dependency relations between signals [59].

• If a process P has n sub-processes, then the subclock ordering relations of the sub-processes intersect at most $n-1$ times.

The intersections between subclock orderings may happen due to computation of shared variables between the different subprocesses. The shared variables represent interaction between subprocesses. Each such intersection results in spawning another sub-process. We represent such sub-process of two sub-processes P_i and P_j as $P_{i,j}$, such that $P_{i,j} \subset P_i, P_{i,j} \subset P_j$ and $P_{i,j}$ has a master trigger. For example: In Figure 12(a), the computation of x, y, w can be considered as 3 independent sub-processes P_x, P_y, P_w with $\hat{x}, \hat{y}, \hat{w}$ as master triggers respectively for each sub-process. P_x is the upper part of the process that reads v and outputs the values of x when $x \geq 10$. P_y is the dual of P_x for u and y and P_w is simply the process implementing the $+$ actor on the right. The clock tree (subclock relation) of P_w is a result of intersection of clock trees for P_x and P_y. Scheduling such processes requires synchronization constraints and we have to ensure that the schedule does not result in a deadlock.

If the model satisfies all the above conditions, then the resulting $\{x_1,..x_n\}$ is the set of partial triggers for P. Using these partial triggers and the clock trees for the sub-processes, we can generate multi-threaded code by the mapping technique explained in Section 3.4.

3.3.2 Computing Partial Triggers

Let B_p represent the system of Boolean equations derived from all the actors present in P. Computing partial triggers for model P is effectively computing prime implicate (with positive literals only) for the CNF formula constructed using B_p. But computing prime implicate considering entire B_p takes a substantial amount of time. We propose a different approach that computes partial triggers almost two orders faster and is shown in Algorithm 1.

```
Algorithm 1: Compute Partial Triggers Set $S = \{x_1, ..x_n\}$
Input: $K$ = Set of all signals in model $P$
Output: Set of partial triggers $S$ for model $P$
Let $S_p$ be set of possible partial triggers
$S_p \leftarrow \{\}$; $S \leftarrow \{\}$;
foreach Signal $y \in K$ do
    if $\nexists z \in K \mid \sigma(z) \supset \sigma(y)$ then
        $S_p \leftarrow S_p \cup \{y\}$;
    end
end
$SubS_p \leftarrow$ all subsets formed using elements of $S_p$
foreach $ele \in SubS_p$ in incremental order of number of signals in ele do
    Set all signals in ele to be absent
    if all signals in $K$ can be deduced to be absent then
        $S \leftarrow ele$;
        return;
    end
    else
        $SubS_p \leftarrow SubS_p - \{ele\}$;
    end
end
```

Let K be set of all signals in model P, S_p be the set of possible partial triggers and S be the minimal set of partial triggers for P. A signal $y \in K$, cannot be a possible partial trigger if $\exists z \in K$, such that $\sigma(z) \supset \sigma(y)$. Let $SubS_p$ contain all nonempty subsets of S_p. We then select each element $ele \in SubS_p$, in the increasing order of the number of signals it contains, arbitrarily breaking ties. We set all signals in ele to be absent and check if this implies that the rest of the signals in K are also absent. This is done by setting all the Boolean variables b_x corresponding to signal $x \in ele$ to false in the Boolean formulae derived. If the only satisfiable assignment for the Boolean formulae is that all variables of the form b_y is false, then we found a set of partial triggers. If no, repeat the procedure with another element of $SubS_p$. At the end of Algorithm 1, S contains the set of partial triggers or the model does not have a concurrent implementation.

The complexity of Algorithm 1 depends on the complexity of the second *for* loop, which is $O(2^n)$, where n is the cardinality of S_p. We use various techniques to keep the cardinality of S_p to be as small as possible and hence Algorithm 1 completes very quickly even though its complexity is $O(2^n)$. This can be witnessed from the experimental results in last part of this Section 3.3.

3.3.3 Constructing the forest of clock trees T

In case of sequential code generation, the clock tree has a single root node which corresponds to the master trigger. The child nodes of this clock tree correspond to the signals whose epochs are subsets of the epochs of the signal above them. For the purpose of understanding, this structure can be thought of as a pyramid, where the top of the pyramid corresponds to master trigger of the process and each level below it corresponds to the signals whose epochs can be directly computed if the epoch of master trigger is known. This levelization is done by repeatedly computing prime implicates of the reduced Boolean formula [43]. This reduced Boolean formula is obtained by setting the boolean variables corresponding to the signals above the current level to true. For example, the signal/s at n^{th} level are obtained by computing prime implicates of the reduced Boolean formulae in which all the boolean variables corresponding to the signals in first $n-1$ levels are true.

In case of multi-threaded code generation, the subclock relation has multiple root nodes which correspond to the partial triggers. The child nodes of the multiple roots are derived by recursive prime implicate generation considering one partial trigger at a time. Figure 13(a) shows a pyramid representation of the clock tree in case of a single master trigger and 13(b) shows the same for multiple partial triggers. Algorithms 2 and 3 build the clock tree. The function $setTrue(x)$ produces a reduced Boolean formula that is further used for prime implicate computation. The function $setFalse(x)$ does the opposite, it sets the variable passed in parameter to $false$. We use it to indirectly select the Boolean formula corresponding to a sub-process: first we set one partial trigger to false ($B_{x=0}$), it marks absence of the partial trigger and all its sub-process, then we complement it ($B_x = B - B_{x=0}$) and get back all sub signals of the partial trigger. The function $PI\_Gen_{SMT}()$, takes a Boolean formula in CNF form and outputs prime implicate. For a smaller Boolean formula, the function $PI\_Gen_{SMT}()$ is quite fast and hence we use it in building parts of clock tree. This function uses [86] SMT solver to generate prime implicates as described in [40].

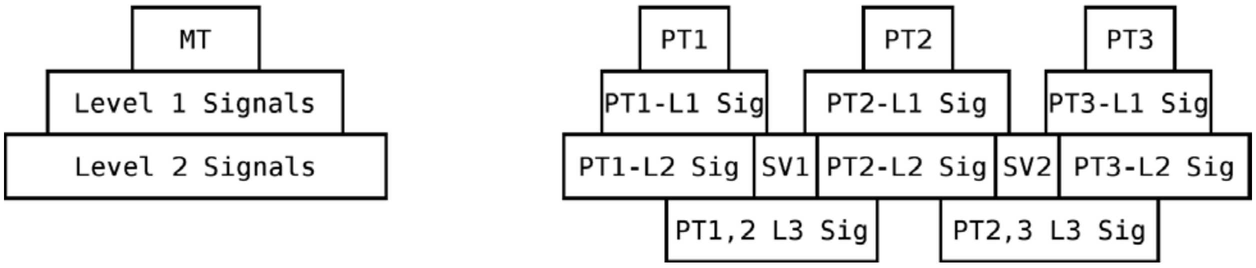


Figure 13: Pyramid structure of clock tree and forest of clock trees for sequential and concurrent specifications

```
Algorithm 2: Compute Clock Tree for model $P$
Input: Set of partial triggers $S = \{x_1, ..x_n\}$ for model $P$,
System of Boolean Equations $B_p$ for model $P$
Output: Entire Clock Tree $T$ for model $P$
Let $N_s$ be a set representing nodes of $T$
Let $E_p$ be a set of all epochs of $P$
$N_s \leftarrow null$;
foreach Epoch $x_i \in E_p$ do
    $N_s$.addNode($x_i$);
end
foreach Partial Trigger $x_i \in E_p$ do
    Let $B_{x_i=0} \leftarrow B_p$.setFalse($x_i$);
    Let $B_{x_i} \leftarrow B_p - B_{x_i=0}$; ▷ Note: $B_{x_i}$ is set of Boolean equations for the subprocess of P
    for which $x_i$ is the root
    $N_s$.recNodeOrder($label_{x_i}, x_i, B_{x_i}$);
end
return
```

3.3.4 Check for Data Dependencies and Deadlock

After constructing the clock tree T for model P, we check for cyclic data dependency issues in T. We also check if there are any deadlocks in P. This is done by traversing each branch of the clock tree and analyzing the constraints. If all checks are completed, we conclude that P is concurrent implementable and proceed for identification of shared epochs.

```
Algorithm 3: recNodeOrder($L, x, B_x$)
Input: $L$: Label of a partial trigger node,
$x$: an epoch of Process P,
$B_x$: set of Boolean equations of for a subprocess of $P$ for which $x$ is the root
Output: Recursively builds the sub-tree under node $x$
Let $x_N \in N_s$ be the node corresponding to epoch $x$
if isLabelEmpty($x_N$) then
    Let $B_{x=1} \leftarrow B_x.setTrue(x)$;
    $\{PI\} \leftarrow PI\_Gen_{SMT}(B_{x=1})$;
    foreach Epoch $x_i \in PI$ do
        Let $B_{x_i=0} \leftarrow B_{x=1}.setFalse(x_i)$;
        $B_{x_i} \leftarrow B_{x=1} - B_{x_i=0}$;
          ▷ Note: $B_{x_i}$ is set of Boolean equations for the subprocess of $P$ for which $x_i$ is the
        root
        $x_N$.addChildNode(recNodeOrder($label_{x_i}, x_i, B_{x_i}$));
    end
end
$x_N$.addLabel($L$);
return $x_N$
```

3.3.5 Identification of Shared Epochs

Often signals with different epochs will be involved in some operation (For Ex: Line 5 of Listing 3.3.5 where x and y have different epochs). In such cases, epochs of involved signals will be subset of epochs of multiple partial triggers (Ex: In Listing 3.3.5, epoch of signal w is subset of epoch of signals x and y). Such signals are said to have shared epochs. Identification of such epochs is important because they correspond to shared variables in software. To compute such shared variables, we need to use synchronization barriers. To identify the signals with shared epochs we use a labeling scheme. Algorithm 3 labels each node in the clock tree with a label that corresponds to the root node under which it is present. All the nodes corresponding to signals that have shared epoch will have multiple labels because they will be under multiple root nodes (For Ex: In Fig 12, node corresponding to epoch of w will have 2 labels - u and v). Rest of the nodes will have single labels.

3.3.6 Mapping and Multi-threaded Code Generation

After establishing concurrent implementability and building clock tree T, we need to create a mapping that can be used for code generation. Algorithms 4&5 give an overview of the code generation procedure.

```
**Algorithm 4:** Code Generation
**Input**: Globals: Model $P$, Clock tree $T$, Set of Partial triggers $S$, Dependency Graph $G$,
        Shared epochs $E$
**Output**: Multi-threaded Code $MT$
**foreach** *epoch* $x_i \in S$ **do**
    Create newthread $th_{x_i}$;
    $th_{x_i} \leftarrow th_{x_i} \bullet \mathrm{recCodeGen}(x_i)$;
**end**
**foreach** *node* $x_i \in E$ **do**
    Create newthread $ths_{x_i}$;
    $ths_{x_i} \leftarrow ths_{x_i} \bullet waitConstraint(labels_{x_i})$;
    $ths_{x_i} \leftarrow ths_{x_i} \bullet \mathrm{recCodeGen}(x_i)$;
    $ths_{x_i} \leftarrow ths_{x_i} \bullet notifyConstraint(labels_{x_i})$;
**end**
$MT \leftarrow$main+codeOf($th_{x_0}+..+ th_{x_n}+ ths_{x_0}+ .. + ths_{x_m}$);
**return**
```

```
Algorithm 5: recCodeGen($x$)
Input: Globals: As Algorithm 4, Current node $x$
Output: Recursively generate code for all child nodes
Let $C$ represent code for $x$ and its child nodes $C \leftarrow exportCode(x)$;
foreach child $ch$ of $x$ do
    if $numLabels(ch) > 1$ then
        $C \leftarrow C \bullet$exportWaitNotifyConstraints($labels_{ch}$); return $C$
    end
    $C \leftarrow C \bullet$ Export code for $ch$ and its dependencies $C \leftarrow C \bullet$recCodeGen($ch$);
end
return $C$
```

T has multiple root nodes with each root node corresponding to a partial trigger. Each of the partial trigger acts as a master trigger for the corresponding sub-processes, which can be handled by a single thread. So we create and associate a thread (th_{x_i}) for each partial trigger. Now we traverse T in a depth first manner. For each node we visit, we check the number of labels ($numLabels()$) and the labels it has. The label indicates under which root node/s the current node is present. If it has a single label, then it indicates that its under one root node (not a shared computation). We export the code for this node and append it to the thread corresponding to the thread pointed by the label (root). Since there are no cyclic data dependencies, we only have to ensure that the input signals to this node are computed before the start of code for the current node. If the node has multiple labels, then it indicates that its a shared computation and we need to wait till the dependencies are computed by other thread/s. We export the $wait-notify$ constraints ($exportWaitNotifyConstraints()$) in the current thread's code and then we handle the shared computation in a different thread. To generate code for the thread handling the shared computation, we start with a $wait$ constraint ($waitConstraint()$) for the synchronization condition, then we proceed traversing the sub-tree in depth first manner, export code as earlier and finally add the $notify$ constraint ($notifyConstraint()$). In this way we generate the code for the complete model.

3.3.7 Experimental Evaluation and Discussions

We evaluated our proposed approach on the benchmarks listed in column 1 of Table 7: Benchmark Suite. These benchmarks exhibit either data parallelism or task parallelism or sometimes both.

Table 7: Benchmark Suite

No.	Benchmark	Summary of the benchmark
1.	Array Addition	Simple data parallel addition. Input is integer arrays of length 10K.
2.	Box Filter	Image processing filter which works by computing the average of-surrounding-pixels. It exhibits both data and task parallelism. Size of test input is 256x256 pixels (can be any size).
3.	Energy Meter	A model of the control system used in any common home energy measurement instrument. It exhibits task parallelism. In our test suite, we run the system for 3 iterations.
4.	Sieve of Eratosthenes	A prime number sieve for finding all prime numbers up to any given limit (10 million in our example). It exhibits both task and data parallelism.
5.	Tennessee Eastman (TE) Plant-wide Industrial Process [20]	TE process is a simplified model of a real-life industrial process consisting of a reactor – separator – re-cycler arrangement. In our test suite, we run the TE system for 1 iteration. Due to the complex nature of the model, it is very hard to manually come up with a multi-threaded implementation and to the best of our knowledge, no multi-threaded implementation exists.

In our evaluation approach, we first manually implemented an efficient C/C++ multi-threaded version of the benchmark using low-level threads. We then modeled the same benchmark in MRICDF and used the tool EmCodeSyn (proposed approach) to generate multi-threaded C++ code. We ensured that the outputs of both versions matched. Finally, we measured the performance of both implementations on a workstation that has 4 Intel Xeon E5405 CPUs with 4GB of memory running Ubuntu 10.10. Performance comparison results are listed in Table 8. Column 4 of Table 8 shows the percentage performance difference between the generated multi-threaded code and hand written multi-threaded code. A negative percentage value indicates that the performance of the generated code is lower than the performance of the hand written multi-threaded code by the corresponding percentage. Experimental results show that the performance of the generated multi-threaded code is almost comparable to the hand-written multi-threaded code. On an average, the generated code for the benchmarks considered is 18.5% slower than the hand written code. On further analysis, we noticed that this performance difference arises due to,

- Generated code uses a lot of templates as the code generator is implemented keeping a generic application in mind.
- Generated code sometimes creates more threads than actually required. The work done by the separate threads could have been merged and done by a single thread. This additional thread creation and destruction overhead also slows down the performance.

Table 8: Experimental Results

Model Name	Manual Multi-Threaded Performance		Generated Multi-Threaded Performance		% Performance Diff. Generated vs Manual	Total Code Generation Time
	LOC	T_{multi} (ms)	LOC	T_{gen} (ms)	(- means Gen. code slower)	(ms)
Array Addition	48	12	195	14	-16.6	428
Box Filter	96	67	212	74	-10.4	1274
Energy Meter	215	17	575	18	-5.8	437
Sieve	56	4722	178	6103	-22.62	1022
TE Process	613	3.5	5947	4.8	-37.1	2350

*LOC stands for Lines of Code, T_{multi}, T_{gen} denotes the execution time of the hand-written multi-threaded and generated multi-threaded code respectively.

Theoretically, the scalability of the tool and the proposed approach can be accurately determined when it is applied on a realistic model of a large embedded system (ex: A satellite system). But, modeling such a large embedded system without knowing all the details of the system is not easy. One can also create a large model by duplicating a smaller model. So, we created larger benchmarks by duplicating (2, 4, 8, 16, 32 times) an existing benchmark. The number of inputs, outputs and actors also got multiplied creating the effect of a large embedded system for all practical purposes. Figure 14 shows the time taken for analysis and code generation for these increasing large models. As the models get bigger, there is a linear increase in the time taken for analysis and code generation.

Our experimental results show that there is a linear increase in the time taken to construct the clock tree (epoch analysis) with a linear increase in the number of actors. There is only a marginal increase in time taken for code generation for larger models as the code generation. The time taken for epoch analysis is significantly larger than code generation time. This is because the complexities of epoch analysis algorithms is much higher than that of code generation algorithms. We also show that computing partial triggers using Algorithm 1 is more than two orders faster than using prime implicate generator.

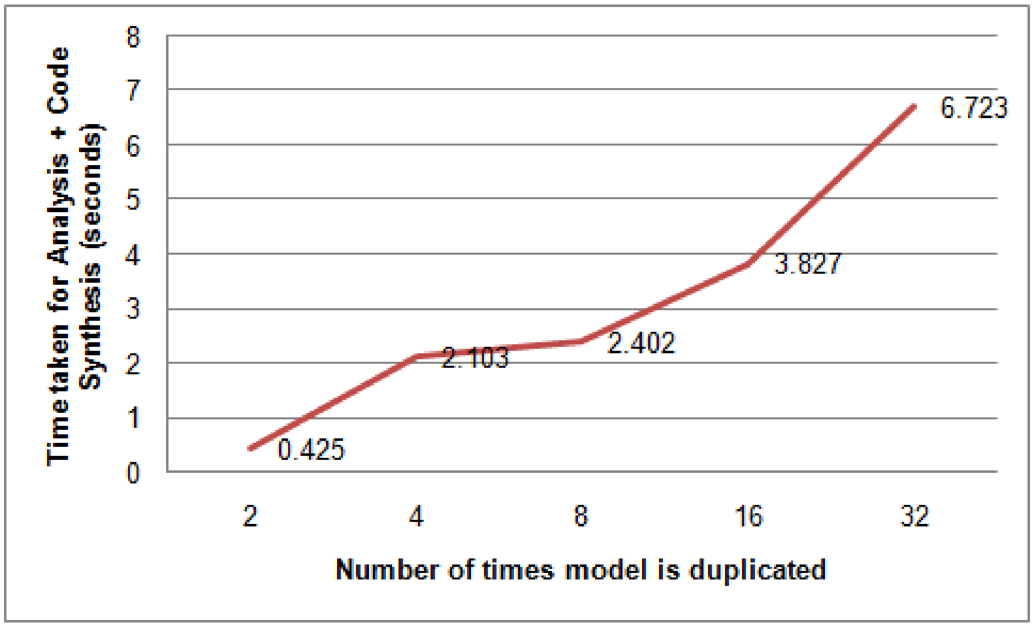


Figure 14: Plot of Time taken for analysis and code generation vs number of times model is duplicated

3.3.8 Summary

Writing concurrent programs, especially for safety critical embedded systems, has always been a error prone task. One of the main reasons for this is – immaturity of concurrent programming models as compared to sequential programming models. In Section 3.3, we explained a correct-by-construction approach for multi-threaded code generation from formal MRICDF specifications. We presented sound techniques to analyze concurrent implementability of MRICDF models and to generate accurate multi-threaded code. Experiments were conducted to compare the performance of the generated multi-threaded code against hand written multi-threaded code. We also conducted experiments to test the scalability of the proposed approach and presented the results. In the current version of the tool, the clock tree construction and the code generator implementation are done targeting accuracy and not efficiency of the generated code. To improve efficiency of the generated code, one can apply optimization transformations on the clock tree which can help in generating load-balanced code. Mapping of partial triggers to threads might not be the most efficient, especially if the amount of work done by the thread is not substantially large than thread creation and destruction overhead. Thus, as another optimization step, one can create a thread pool and map partial triggers to tasks – adopting the concept of Intel Threading Building Blocks (TBB). Excessive and avoidable synchronizations in synthesized multi-threaded code affect its performance drastically. In Section 3.4, we explain an optimization technique that identifies such avoidable synchronizations and improve the efficiency of the synthesized multi-threaded code.

3.4 Synthesis of Improved Multi-threaded Code from Polychronous models based on Analysis of Affine Relations

One of the biggest issues that can effect the performance of multi-threaded code is *Synchronizations*. Synchronization issues are hard to detect and can heavily impact the throughput of the system executing multi-threaded code. While synchronizations are necessary to guarantee the correctness of execution, more often than not, they can be avoided - if not completely, at least partially, without actually affecting the correctness of execution. We name such synchronizations that can be removed and yet not affect the correctness of execution as - "avoidable synchronizations". Identifying avoidable synchronizations manually during design and development phase is very hard, laborious and error prone. Identifying them manually post-development by analyzing the execution traces is hard too. Even if we manage to identify and remove them, we still have to prove the correctness of execution. In a formal model-driven development environment, we can identify such avoidable synchronizations in a more formal way. Further, we can use formal analysis techniques to prove the correctness of execution.

In Section3.3, we proposed a technique to generate multi-threaded code, where the independent threads synchronize by means of barriers. Analysis of this generated code, showed us, that a large amount of execution time is spent in barrier synchronizations, where threads are waiting for other threads to finish their work. This was impacting the performance of the generated multi-threaded code. In Section 3.4, we propose a technique to automatically identify avoidable synchronizations and remove them from the generated code. Our technique is based on analysis of affine relations between the clocks of synchronizing threads. We further conduct experiments to analyze the efficiency of our proposed approach.

3.4.1 Avoidable Synchronizations

Consider a simple application that accepts two independent streaming inputs and produces one streaming output. The application consists of 3 processes *A, B & C*. Processes *A* and *B* transforms the inputs and outputs tokens A and B. Process *C* waits till it receives both tokens and then computes the output. Data flow model of the application is shown in Figure 15.

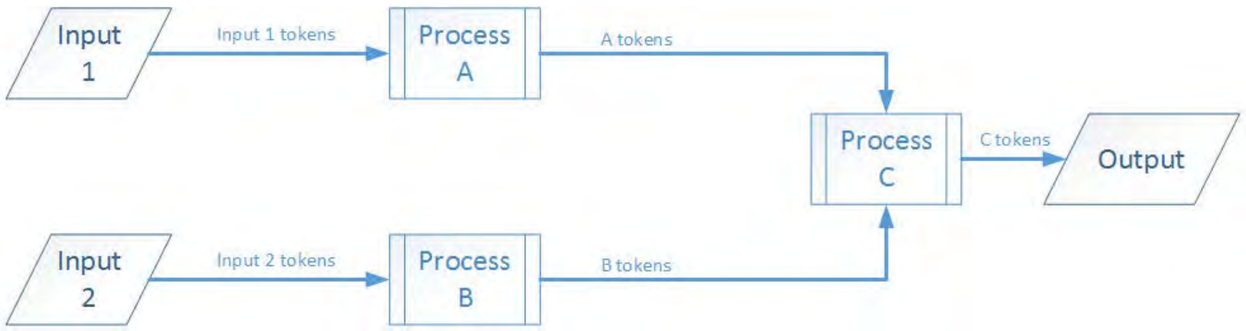
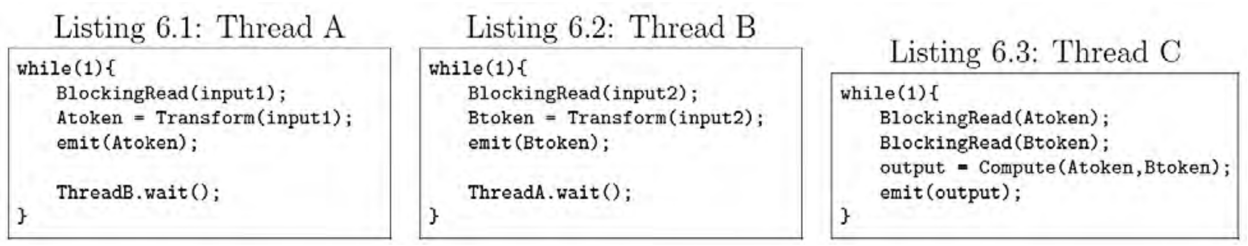


Figure 15: Simple application to illustrate avoidable synchronizations

One can easily implement the logic of this application using 3 threads - Threads T_A, T_B & T_C implementing the logic of Processes *A, B & C* respectively. Thread T_C waits for threads T_A & T_B to finish their work and then it computes the output. Since the inputs are not synchronous, threads T_A and T_B have to also synchronize with each other. It means, the once thread T_A finishes transforming its current input, it has to wait for thread T_B to finish its work before proceeding to the next input. Pseudo-code for this implementation is shown in Figure 16. If no more information regarding the temporal properties of inputs are provided, then all of the synchronizations are essential. If we were provided with additional relations between occurrences of inputs 1 & 2, then we can check to see if all of the synchronizations are necessary or if some of them can be avoided.



Listing 6.1: Thread A

```
while(1){
    BlockingRead(input1);
    Atoken = Transform(input1);
    emit(Atoken);

    ThreadB.wait();
}
```

Listing 6.2: Thread B

```
while(1){
    BlockingRead(input2);
    Btoken = Transform(input2);
    emit(Btoken);

    ThreadA.wait();
}
```

Listing 6.3: Thread C

```
while(1){
    BlockingRead(Atoken);
    BlockingRead(Btoken);
    output = Compute(Atoken,Btoken);
    emit(output);
}
```

Figure 16: Pseudo-code for the application in Figure 15

Assume that, we are given information that between every two successive occurrences of Input 1, there is always an occurrence of Input 2. This also implies that, between every two consecutive outputs of Thread T_A, Thread T_B produces an output too. An example execution trace that is running code in Figure 16 and that satisfies these constraints is shown in Figure 17. In this execution trace, we assume that the time taken by threads to complete their work is almost negligible.

From the trace, we can see that *Input 1* is available every 3 ticks of base clock (t) starting from $t=1$. Hence, thread T_A produces "$A\,Token$" at $t=1,4,7,etc$. On the other hand, thread T_B gets *Input 2* every 3 ticks of base clock (t), but starting from $t=3$. Hence, thread T_B produces "$B\,Token$" at $t=3,6,9,etc$. After thread T_A produces its token, it has to wait till thread T_B to release its token. This wait is to ensure that thread T_A doesn't produce 2 consecutive tokens until thread T_B has produced one too. But, we are guaranteed that this will never happen. Thus, in such cases, we can say that T_A and T_B need not synchronize and the synchronizations between them could be avoided. Pseudo-code for the simple application with reduced number of synchronizations is shown in Figure 18.

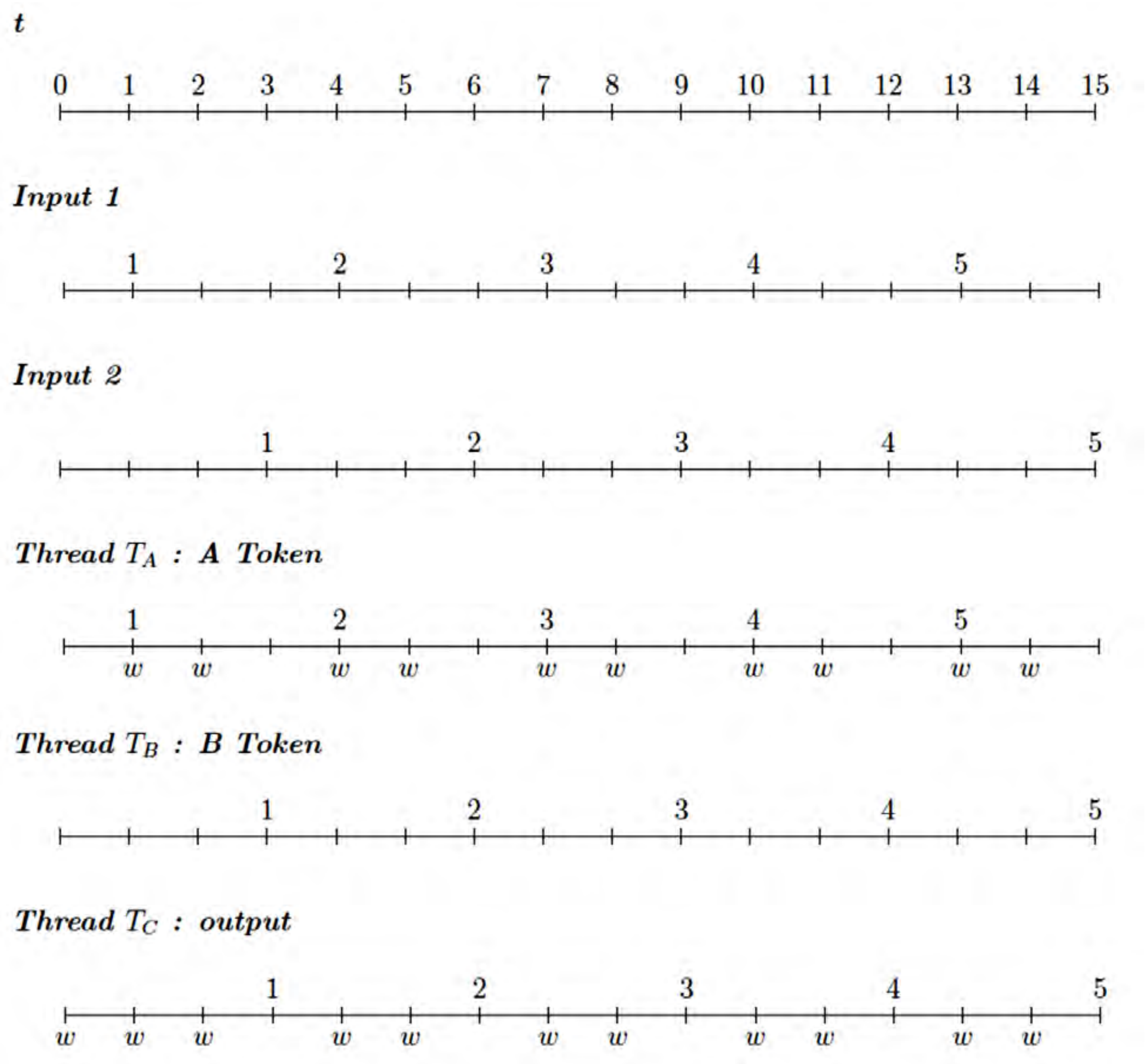


Figure 17: Sample execution trace of the application in Figure 15

Listing 6.4: Thread A

```
while(1){
    BlockingRead(input1);
    Atoken = Transform(input1);
    emit(Atoken);
}
```

Listing 6.5: Thread B

```
while(1){
    BlockingRead(input2);
    Btoken = Transform(input2);
    emit(Btoken);
}
```

Listing 6.6: Thread C

```
while(1){
    BlockingRead(Atoken);
    BlockingRead(Btoken);
    output = Compute(Atoken,Btoken);
    emit(output);
}
```

Figure 18: Optimized pseudo-code for the application in Figure 15

As seen from the above example, identifying such avoidable synchronizations manually even for a trivial application is very laborious and error prone task. An automated way of doing this would be very useful. This is the problem, which we have tried to address in Section 3.4.

Contributions detailed in Section 3.4:

• Based on analysis of affine relations between clocks of synchronizing threads, a novel technique to identify if the threads really need to synchronize or it can be avoided
• Synthesis of efficient multi-threaded code by reducing the number of synchronizations and further improving the throughput

3.4.2 Affine Transformations and Affine Relations in Polychronous Languages

In general terms, an "affine transformation" is a transformation that maps variables into new variables by applying a linear combination of translation, rotation, scaling and/or shearing. The authors of [73], first proposed the concept of affine transformations of clocks in Polychronous language - Signal. In their initial work, the authors explained how to formally express a clock as affine transformation of another clock. These affine transformations induce affine relations between the clocks, which help in deriving new set of synchronizablity rules. The authors further extended their initial work of [73] to [72], where they proposed an augmented clock calculus technique that accepts Signal specifications and affine clock relations, analyzes it and synthesizes code for real-time systems.

3.4.2.1 Affine Transformations and Relations

An affine transformation of a clock is expressed in terms of 3 parameters, namely - n, ϕ and d. An (n,ϕ,d) affine transformation when applied on a clock H produces another clock K by inserting $(n-1)$ instants between any two successive instants of H and counting each d^{th} instant, starting from ϕ^{th} instant of H. An example is shown in Figure 20. A clock more frequent than H is derived from clock H by inserting $(5-1)$ instants between any two successive instants of clock H. Clock K can then be derived from this frequent clock, by counting every 9^{th} instant, starting from 4^{th} instant. In Figure 19, clock H and K are represented using circles, while the frequent clock is indicated using the vertical lines.

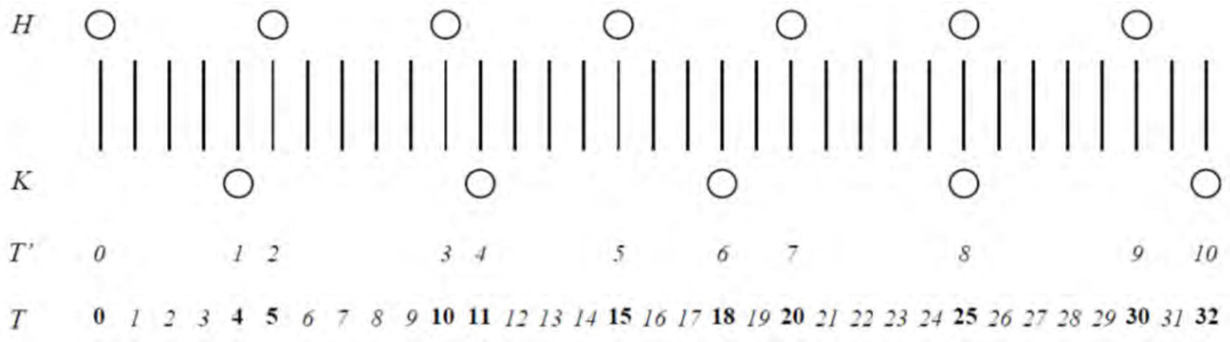


Figure 19: Clock K is (5; 4; 9) affine transformation of clock H

(n,ϕ,d) affine transformation defines an (n,ϕ,d)-affine relation between the corresponding clocks and is denoted as $H \xrightarrow{(n,\phi,d)} K$.

Given, a (n,ϕ,d)-affine relation between H and K, there also exists a $(d,-\phi,n)$-affine relation between K and H. In other words, $H \xrightarrow{(n,\phi,d)} K$ implies $K \xrightarrow{(d,-\phi,n)} H$. In [73] and [72], the authors have explored properties of affine relations such as equivalence, composition, etc. Further, they show how the properties of affine relations can be used to derive new synchronization constraints for the model.

3.4.2.2 Constructs to express Affine Transformations and Relations

To express affine transformations in Polychronous language Signal, the authors of [73], proposed 3 new operators - *sample{ϕ,d}(H), unsample{n,ϕ}(K)* and *clk_affine{n,ϕ,d}(H, K)*. We present here a brief description of these new operators. For a detailed overview, we direct the readers to [73].

- *Y = sample{ϕ,d}(X),* ϕ and d positive

The output of this operator Y, is a down-sample of X with a period d and phase ϕ.

- *clk_affine{n,ϕ,d}(H, K)*

This operator defines the an (n,ϕ,d)-affine relation between H and K. This is defined in terms of sample operator as,

$$\begin{aligned} clk\_affine\{n,\phi,d\}(X,Y) = \quad & (|\, H := sample\{max(0,-\phi),n\}(I) \\ & |\, K := sample\{max(0,\phi),d\}(I) \\ & |\, H \ \hat{}{=}\ X \\ & |\, K \ \hat{}{=}\ Y \\ & |) \\ & where\ n,d > 0, \phi \in Z \end{aligned}$$

- *Y = unsample{n,ϕ}(X, Z)*

The output of this operator Y, is an affine over-sampling of X using Z. This is defined as,

$$\begin{aligned} unsample\{n,\phi\}(X,Z) = \quad & (|\, clk\_affine\{n,\phi,1\}(X,Y) \\ & |\, Y := (X\ when\ \hat{}Y)\ default\ Z \\ & |\, Y \ \hat{}{=}\ Z \\ & |) \\ & where\ n > 0, \phi \in Z \end{aligned}$$

3.4.3 Analysis of Affine Relations for Improved Multi-threaded Code Synthesis

In Section 3.3, we proposed the concept of partial triggers and defined the constraints a processes has to satisfy for it to be concurrently implementable. Recollecting from Section 3.3, a process that can be partitioned into multiple sub-processes is concurrently scheduled, if we can identify a set of partial triggers, where each partial trigger acts as master trigger for each of the sub-process. The synchronization between sub-processes is captured by the intersection between the clocks of partial trigger signals. If we are given an affine relation between the clocks of two partial trigger signals, then we can analyze the affine relation and derive additional synchronization constraints between the clocks of the two partial trigger signals. Furthermore, these additional synchronization constraints can help in determining if the synchronization between the corresponding sub-processes is essential or it can be avoided. This is the main contribution explained in Section 3.4. To illustrate the idea, let us consider a simple process with three sub-processes as shown in Figure 21. Process 3 waits for completion of Process 1 and 2 before executing and process 1 and 2 synchronize via a barrier. The hierarchical clock graph is shown in Figure 20. H, K and W are the partial triggers for the three processes 1, 2 and 3 respectively. With this background, an interesting question is - Do Process 1 and 2 have to synchronize always? If no more information is provided with regards to the relation between H, K and W, then Process 1 and 2 have to synchronize always.

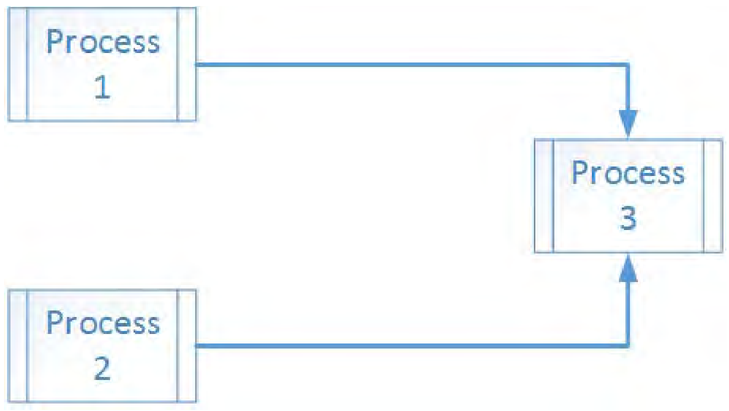


Figure 20: Hierarchical Clock Graph

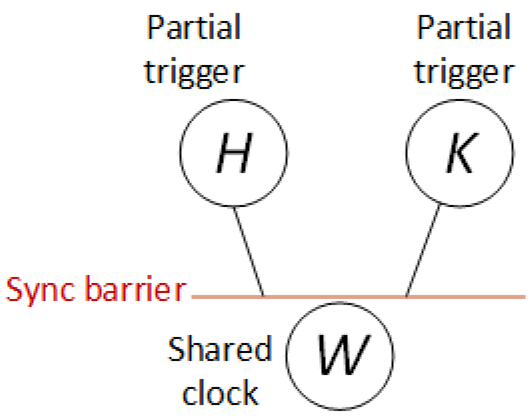


Figure 21: Simple Process with 3 sub-processes

Let us say, there exists an affine relation between H and K, in other words $H \xrightarrow{(n,\phi,d)} K$ exists. Depending on values of n, ϕ and d, we now explore to see if Process 1 and 2 need to synchronize or not.

Case 1: $H \xrightarrow{(n,\phi,d)} K$, $n = d$, $\phi = 0$

This is a trivial case. When $n = d$ and $\phi = 0$, clock H and K are synchronous with each other. They need not be synchronized. In fact, with this constraint, the process becomes endochronous and it need not even be implemented using multiple threads. An example execution trace with these constraints is shown in Figure 22.

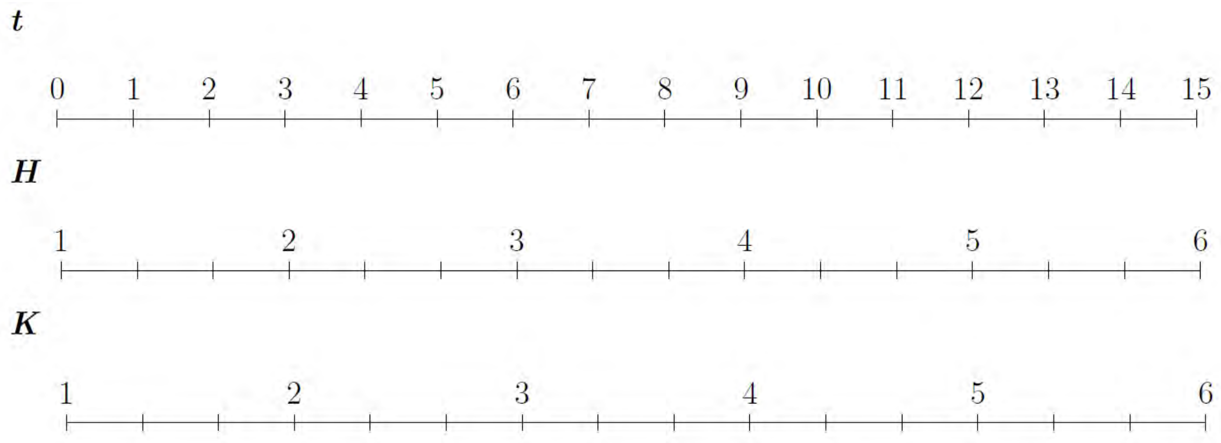


Figure 22: Example execution trace of the application in Figure 21 when $n = d = 3$, $\phi = 0$

Case 2: $H \xrightarrow{(n,\phi,d)} K$, $n = d$, $\phi \neq 0$, $\phi \leq n$

Under this condition, we are guaranteed that between any two occurrences of H, there is always an occurrence of K. Under such constraints, Processes 1 and 2 need not synchronize. An example execution trace with these constraints is shown in Figure 23.

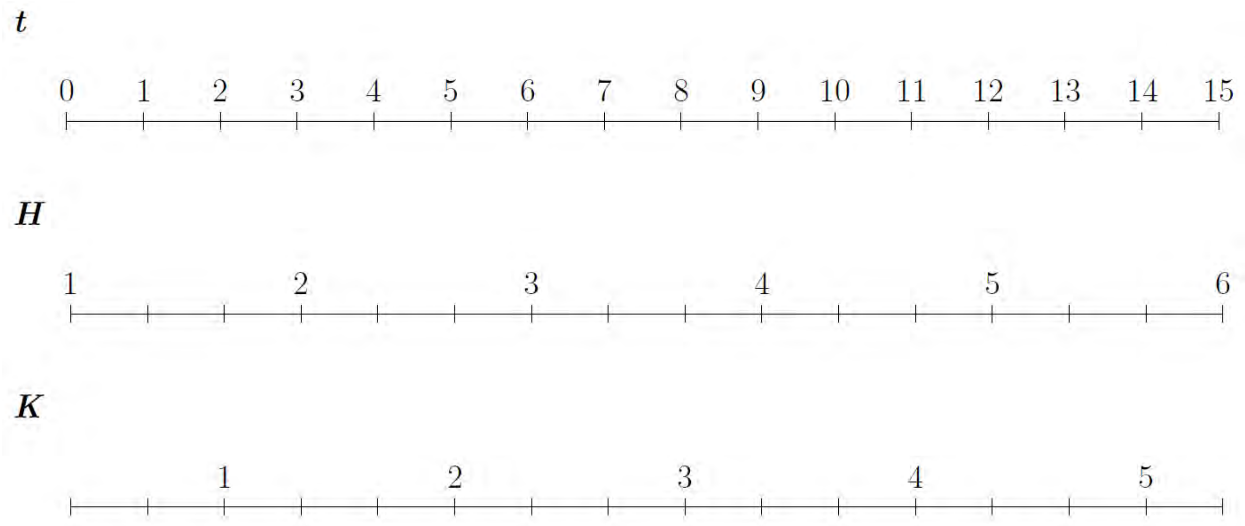


Figure 23: Example execution trace of the application in Figure 21 when $n = d = 3$, $\phi = 2$

Case 3: $H \xrightarrow{(n,\phi,d)} K$, $n = d$, $\phi \neq 0$, $\phi > n$

For these constraints, we are guaranteed that between any two occurrences of H, there is always an occurrence of K, but not for the first ϕ instants. To avoid synchronization between the two processes, we need to introduce a buffer to store the outputs produced from Process 1 during first ϕ instants. The size of the buffer to be introduced is $ceil(\frac{\phi}{n})$. An example execution trace with these constraints is shown in Figure 24.

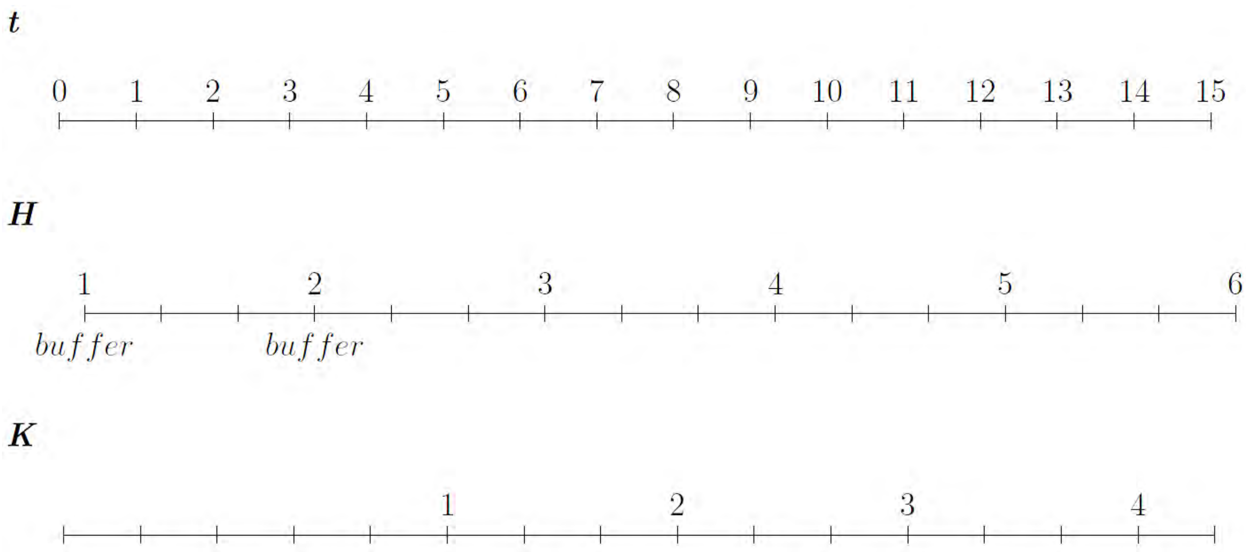


Figure 24: Example execution trace of the application in Figure 21 when $n = d = 3$, $\phi = 5$

Case 4: $H \xrightarrow{(n,\phi,d)} K$, $n \neq d$, $\phi = 0$

When $n \neq q$, the synchronization between Process 1 and 2 cannot be avoided completely. One process is producing outputs at a higher rate as compared to other process. In such cases, instead of synchronizing every instant, we could introduce buffers and then synchronize only when the buffer is full. The size of the buffer depends on the value of n and d. If $n < d$, then a buffer of $ciel(\frac{d}{n})$ is added to store output of the process 1 (process whose partial trigger is H). If $n > d$, then a buffer of $ciel(\frac{n}{d})$ is added to store output of the process 2 (process whose partial trigger is K). During execution, the threads for these processes check if the buffer is full or not. If it is full, then synchronize with the other process by waiting. If it is not full, then they don't need to synchronize. An example execution trace with these constraints is shown in Figure 25.

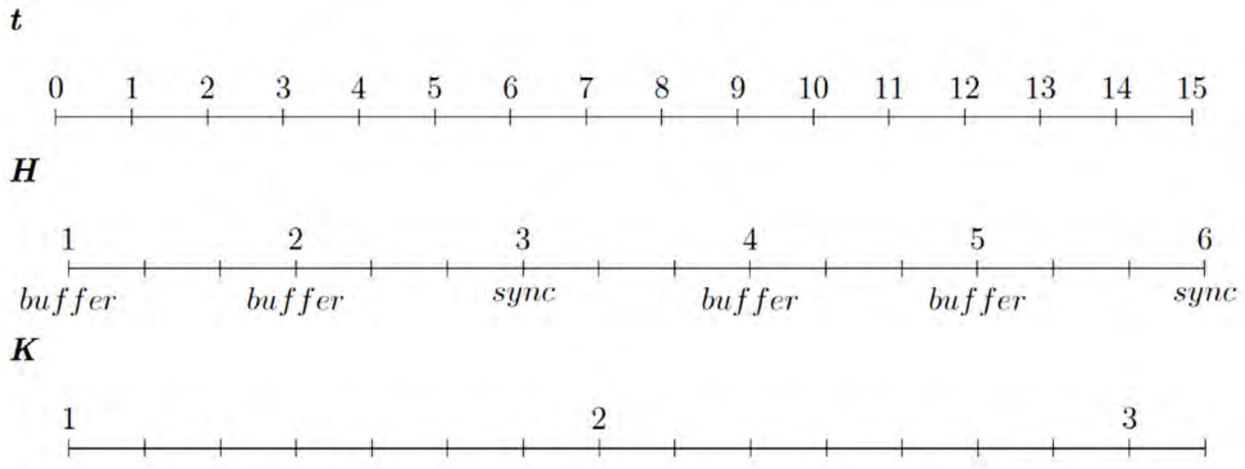


Figure 25: Example execution trace of the application in Figure 21 when $n = 3, d = 7$, $\phi = 0$

Case 5: $H \xrightarrow{(n,\phi,d)} K$, $n \neq d$, $\phi \neq 0$

This case is very similar to case 4 except that $\phi \neq 0$. We now have to increase the buffer size computed in case 4, with $ciel(\frac{\phi}{n})$ if $n < d$ or $ciel(\frac{\phi}{d})$ if $n > d$ An example execution trace with these constraints is shown in Figure 26.

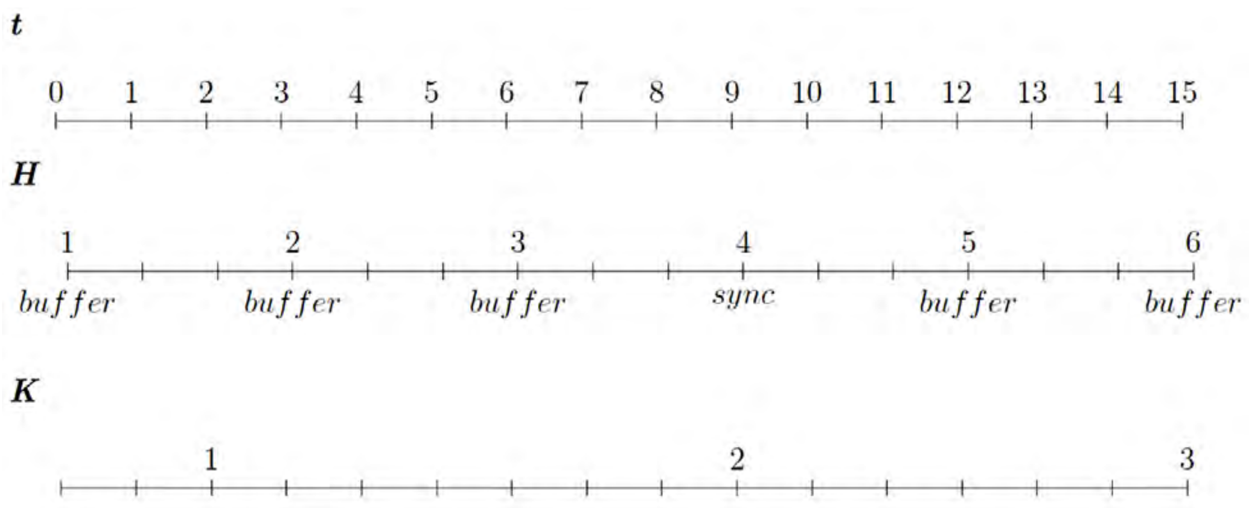


Figure 26: Example execution trace of the application in Figure 21 when $n = 3, d = 7$, $\phi = 2$

Thus, given an affine relation between two synchronizing clocks, by analyzing the n, ϕ and d values we can automatically identify if synchronization is necessary or it can be avoided.

3.4.4 Summary

In Section 3.4, we proposed a fully automated approach to identify avoidable synchronizations based on the analysis of affine relations between two synchronizing clocks. Using an example, we showed how identification of avoidable synchronizations results in synthesizing multi-threaded code with reduced number of $wait()$ statements (barrier synchronizations). Proposed approach being based on formally sound techniques, proving the correctness of execution can also be done formally. One of the major limitations of the proposed approach is that, affine relations between clocks of interest do not always exist. Even if they do exist, there is no automated way to infer from the given specifications/model. The user would have to manually enter these relations before clock calculus step. Our proposed approach is limited to handle affine relations between 2 synchronizing clocks. But, more often than not, more than 2 signals will participate in barrier synchronizations. Analysis and deriving implicit synchronization constraints considering affine relations between more than 2 clocks is difficult.

3.5 Synthesis of Application-Specific Instruction-set Processor(ASIP) from Polychronous models

Ever increasing performance requirements of hardware platforms have motivated the designers to explore application specific processors with custom instruction sets. In [55], the authors propose a formal graph model, called *Conditional Partial Order Graphs (CPOG)* as a semantic model for describing the semantics of individual instructions, and use that to synthesize control and data-path implementing the instruction set. CPOGs provide compact representations of partially ordered sets where the orderings are often conditioned on predicates and not fixed. The work explained in Section 3.5, assumes the fact that when a designer wants to off-load a specific computation intensive function onto a co-processor, he/she can actually describe the function in a high level language, and should be able to synthesize the processor, and its instruction set. Also, describing custom instruction functionalities in terms of partial orders may not be convenient for designers as they have to first conceive the instruction set, and its semantics and then implement the computation in that instruction set. Therefore, in Section 3.5 we explain how to extend the work in [55] and propose an approach by which formal MRICDF/SIGNAL [43] specifications can be compiled to CPOGs which can further be used to generate ASIPs[1]. The reason for choosing MRICDF/SIGNAL, even though they were invented for synthesizing control software, is that they are data-flow languages, but unlike other languages in the synchronous family, these are polychronous[33] in nature. The polychronous model of computation allows data-flow to progress asynchronously whenever possible unless they need to synchronize to share certain data processing. Thus, concurrency is well captured and the control state machine that exploits the concurrency for performance with component reuse can be easily synthesized as we show here.

3.5.1 Conditional Partial Order Graphs

3.5.1.1 Definition: Conditional Partial Order Graph

A CPOG [56] is a quintuple $G = \langle V, E, X, \rho, \phi \rangle$ where,

• V is a set of *nodes* which corresponds to events/atomic actions in a system that is being modeled.

• $E \subset V \times V$ is a set of directed *edges* between the *nodes*. The direction of the edges indicate the dependencies between the events/atomic actions. An edge from node n to node m, indicates m depends on n.

• X is a set of Boolean variables. Each of these individual Boolean variables could be assigned values $\{0,1\}$ resulting in unique 2^n possible codes with n bit words.

• ρ is a *restriction function* defined on the set of Boolean variables in X as $\rho \in \mathsf{F}(X)$, where $\mathsf{F}(X)$ is the set of all Boolean functions on the Boolean variables in X. ρ basically defines the operational domain of the CPOG. Of the 2^n possible codes obtained by assigning $\{0,1\}$ to each variable in X, only those which satisfy ρ are valid.

1 Application Specific Instruction set Processor

- ϕ is a function such that, $\phi : (V \cup E) \rightarrow \mathsf{F}(X)$. It assigns a Boolean condition $\phi(z)$ to every node and edge z in the graph G.

Diagrammatically, CPOGs can be represented as directed graphs.

Now we show how CPOGs can be used to encode semantics of instruction sets following the idea described in [55]. Consider a simple adder/subtractor application which does *add* or *subtract* depending on a $select$ signal. Expressing the application in terms of atomic instructions and assigning a unique name for each instruction, we get the following table.

Adder(A=A+B; $select$)	**Subtractor(A=A-B; $\overline{select}$)**
I_1**:Load A**	I_1:Load A
I_2**:Load B**	I_2:Load B
I_3**:Compute A+B**	I_5:Compute A-B
I_4**:Store A**	I_4:Store A

To represent these two instructions functionalities with CPOG, we first create a graph H with 5 nodes and 6 edges. The nodes represent the atomic instructions and the edges represents the dependencies between them. Dependency between atomic actions means data produced by one action is used by another action. Conditional dependencies are represented using annotations on the edges. Pictorially, this is shown in Figure 27(A). Nodes I_1, I_2, I_4, all have $\phi = 1$ (unconditional), while node I_3 has $\phi(I_3) = select$ and node I_5 has $\phi(I_5) = \overline{select}$, both conditional. All edges are conditionals. Figure 27(A) represents the above mentioned adder/subtractor instruction set as a CPOG, the projections Figure 27(B) and 27(C) represent the behavior of the modelled system under the constraints, operation variable $select = true$ and $select = false$ respectively. The greyed nodes and edges in Figure 27(B) and 27(C) indicate that the corresponding nodes/edges do not contribute to that particular behavior of the system.

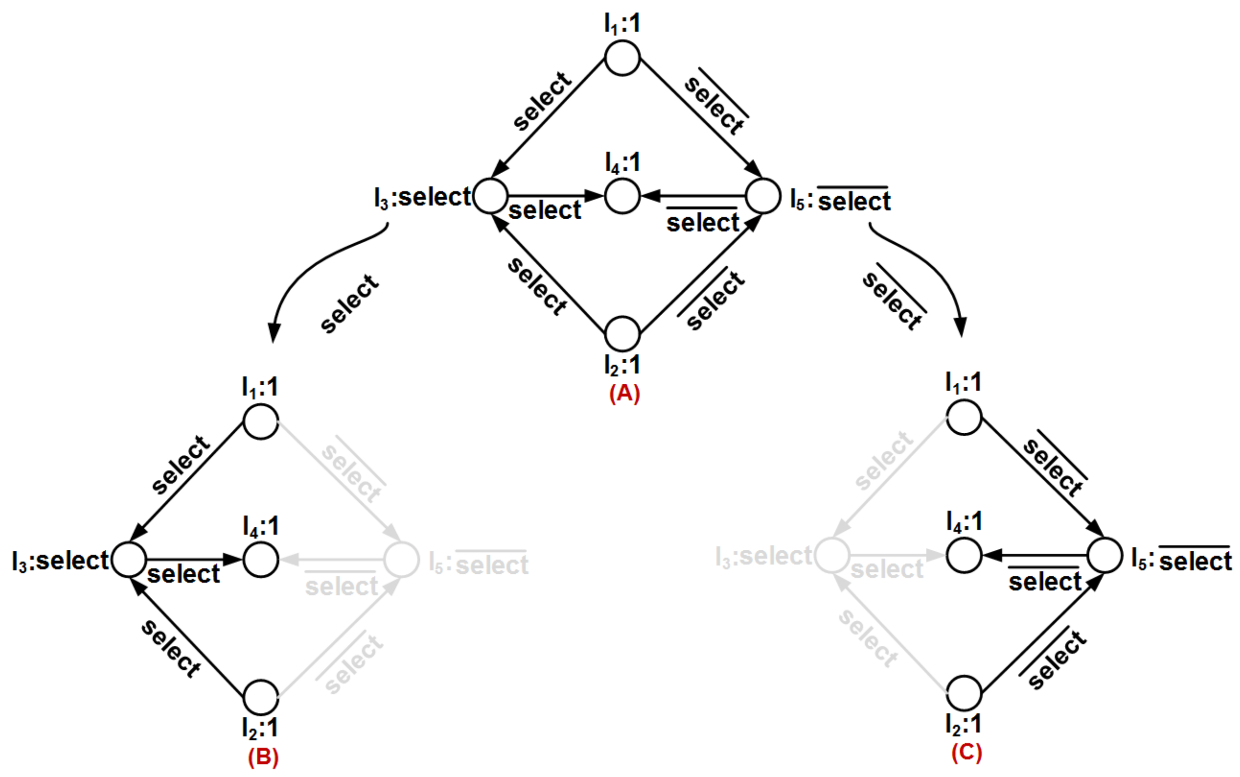


Figure 27: Graphical representation of CPOG

(A) Graphical representation of CPOG H, (B) $H|_{select}$, (C) $H|_{\overline{select}}$

One can notice from the Table and Figure 27, that atomic actions I_1 and I_2 can be executed either concurrently or sequentially in any order as long as it is executed before the instruction that is dependent on it. There exists a partial order on the set of atomic actions. Further, the operational vector X contains set of Boolean variables for which values $\{0,1\}$ could be assigned. Thus, for each of the partial order, there exists a unique Boolean vector. In this example, the cardinality of operational vector $|X|=1$ and the vector is $X = \langle select \rangle$. One can use these Boolean vectors as opcodes for instructions and an instruction set with unique opcodes could be constructed. Composition of two instruction sets which don't share common opcodes is defined as the union($\cup$) of them. If there are multiple instruction sets, then their composition is done by doing pairwise composition. For further details on composition we refer the readers to [56].

3.5.2 MRICDF Actors and their CPOGs

Recall that the nodes in CPOG represent events/atomic actions and the edges represent the dependencies between them. In MRICDF model of computation, during any reaction, computation of signal values are atomic actions and hence can be represented using nodes of a CPOG and the data dependency between signals can be indicated using the edges of a CPOG. With this idea, we now explain how to derive CPOGs representing the control and scheduling information for MRICDF actors.

Function: A function actor performs any user specified state-less synchronous reaction. The input and output signals of the function actor participate in the exact same set of reactions. A state machine is not a function actor, because it has state. The generic definition of a Function actor is as below,

Operation: $y = f(x_1, x_2, ..., x_n)$

Clock relation: $\hat{y} = \hat{x}_1 = \hat{x}_2 = ... = \hat{x}_n$

- In the above definition $y, x_1, x_2, ..., x_n$ are signals. Thus we have,

$$V = \{y, x_1, x_2, ..., x_n\}$$

- Output signal y depends on all of the input signals $x_1, x_2, ..., x_n$;

$$E = \{x_i \to y \mid x_i \in (x_1, x_2, ..., x_n)\}$$

- The Boolean variable set X consists of variables that represent the clocks of all the signals. The Boolean variable is true, if the corresponding clock is present, else it is false. Thus, for a Function actor we have,

$$X = \{\{b_y\} \cup \{b_{xi} \mid x_i \in (x_1, x_2, ..., x_n)\}\}$$

- The restriction function ρ defines the values that Boolean variables in X can take. For a Function actor, we know that clocks are all synchronous. Hence ρ is represented by the set of clauses short handed as,

$$b_y = b_{x1} = b_{x2} = ... = b_{xn}$$

- Function ϕ, assigns Boolean condition for each of the signals and the dependencies. Each of the signal is present only if its clock is present and the output is dependent on each of the input signals.

For nodes we have,

$$\phi(y) = b_y$$
$$\phi(x_1) = b_{x1}$$
$$...$$
$$\phi(x_n) = b_{xn}$$

and for edges we have,

$$\phi(x_1 \to y) = b_{x1}$$
$$\phi(x_2 \to y) = b_{x2}$$
$$\dots$$
$$\phi(x_n \to y) = b_{xn}$$

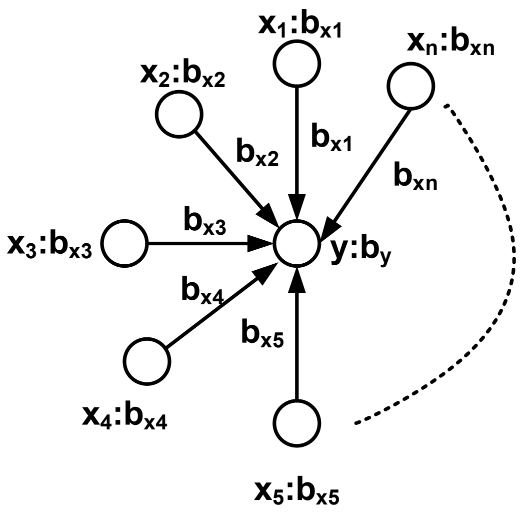


Figure 28: CPOG for Function Actor

Figure 28 shows the graphical representation of the CPOG for Function actor.

• **Buffer:** The Buffer actor temporarily stores the value of a signal arriving at its input port for the next reaction. In other words – it delays the signal. The next reaction occurs when a new value appears at its input. The value stored from the previous reaction is sent to the output during the current reaction. Buffers are synchronous actors. Buffers can be cascaded to store data across multiple subsequent reactions – creating multiple delay lines.

Operation: $y = x \ \$ \ 1\, init\ c$

Clock relation: $\hat{y} = \hat{x}$

- In a given reaction, y, x are the only two signals of a Buffer actor, c is just a constant value. Hence,

$$V = \{y, x\}$$

- There are no dependencies between signals in a Buffer actor. Hence $E = \{\}$

- There are 2 clocks for a Buffer actor and hence 2 Boolean variables. Thus we have,

$$X = \{b_y, b_x\}$$

- Similar to Function actor, the clocks of both the signals of Buffer actor are synchronous. Thus we have ρ as,

$$b_y = b_x$$

- Both signals are present only if their clocks are also present and there are no dependencies between signals in Buffer actor. Thus we have,

$$\phi(y) = b_y$$

$$\phi(x) = b_x$$

Figure 29 shows the graphical representation of the CPOG for Buffer actor.

Figure 29: CPOG for Buffer Actor

• **Sampler:** A Sampler actor samples the input signal on the first input port based on a Boolean condition that occurs on second input port. A sampler does not react unless it has value on both its inputs, and the second input value must be *true*. The set of reactions in which the output is written is the intersection of the set of reactions in which the first input participates, and the set of reactions when the second input is *true*. Thus sampler is a *polychronous* actor.

Operation: $y = x \; when \; c$
Clock relation: $\hat{y} = \hat{x} * [c]$

- In a sampler actor, along with the signals $y, x \;\&\; c$, two other synthesized signals $[c] \;\&\; [\bar{c}]$ also exist. These signals in the physical world represent signal c when $true$ or $false$ respectively.

$$V = \{y, x, c\}$$

- The output of a sampler actor depends on both, input and the condition. Thus we have 2 dependencies which result in 2 edges as defined below,

$$E = \{x \to y, c \to y\}$$

- Including the clocks for the synthesized signals, we have 5 different clocks and thus we have 5 Boolean variables listed below,

$$X = \{b_y, b_x, b_c, b_{[c]}, b_{[\bar{c}]}\}$$

- From the clock relation for Sampler actor and the definition of synthesized signals we have

$$\rho = \begin{Bmatrix} \{b_y = b_x \wedge b_{[c]}\} \cup \\ \{b_c = b_{[c]} \vee b_{[\bar{c}]}\} \cup \\ \{b_{[c]} \wedge b_{[\bar{c}]} = \mathit{false}\} \end{Bmatrix}$$

- The output signal clock is present only if the input signal is present and the condition is *true* . Hence we have,

$$\phi(y) = b_y$$
$$\phi(x) = b_x$$
$$\phi(c) = b_c$$

$$\phi(x \to y) = b_x \wedge b_{[c]}$$
$$\phi(c \to y) = b_x \wedge b_{[c]}$$

Figure 30 shows the graphical representation of the CPOG for Sampler actor.

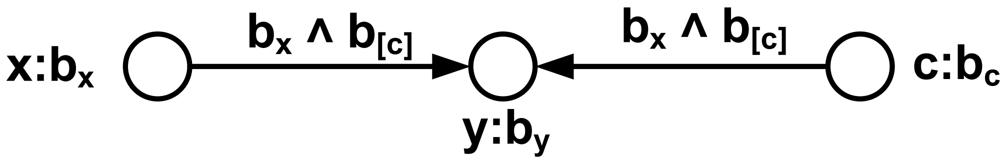


Figure 30: CPOG for Sampler Actor

• **Merge:** A merge actor merges two input signals with same/different clocks to produce an output. During the merge, higher priority is given for the signal on the first input, i.e, when input occurs on just the first input signal, or when both input signals have values, the value on the first input is passed onto the output signal. The clock of the output signal is the union of the clocks of the input signals. Merge is therefore another polychronous actor.

Operation: $y = x \textit{ default } z$

Clock relation: $\hat{y} = \hat{x} + \hat{z}$

- The merge actor has 3 signals and hence we have 3 nodes.

$$V = \{y, x, z\}$$

- The output of merge actor y, depends on first input x, if it is present, else it depends on second input y. Thus we have 2 dependencies which results in 2 edges as defined below,

$$E = \{x \to y, z \to y\}$$

- The merge actor has 3 signals and hence we have 3 Boolean variables.

$$X = \{b_y, b_x, b_z\}$$

- From the clock relation for merge actor we have,

$$\rho = \{b_y = b_x \vee b_z\}$$

- For a merge actor, the output signal clock if either of the inputs is present and is absent when neither is present. Hence we have,

$$\phi(y) = b_y$$
$$\phi(x) = b_x$$
$$\phi(z) = b_z$$

$$\phi(x \to y) = b_x$$
$$\phi(z \to y) = b_z \wedge \overline{b}_x$$

Figure 31 shows the graphical representation of the CPOG for Merge actor.

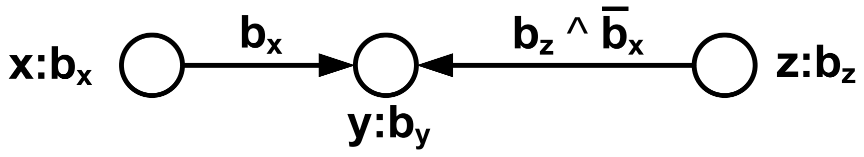


Figure 31: CPOG for Merge Actor

Proposition 1 *For each primitive actor A, if g_A represents the CPOG derived using the steps described above, then g_A contains all the necessary information for control of scheduling the execution of A.*

Proof: By definition.

Proposition 2 *For primitive actors A_1 and A_2, if g_{A_1} and g_{A_2} represents the corresponding CPOGs then for composition $A_1 \mid A_2$, the corresponding CPOG is the union of g_{A_1} and g_{A_2}.*

Proof: Two actors can be composed if they are compatible, and the union($\cup$) of CPOGs have the same compatibility test. The union($\cup$) of CPOGs is followed by a compatibility test which tests for contradicting clock relations, and hence if the union exists, it provides the control of scheduling of the individual actors during execution of the composition.

- **Composite Actor:** Composite actors are a combination of primitive actors that are used to express modular and hierarchical behavior. In order to derive the CPOG of the whole model, it is essential to first derive the CPOGs of composite actors and then compose ($\cup$) it with the CPOG of the rest of the model. One can define composite actors using structural induction. Algorithm 6[2] lists the method used to derive a CPOG for a composite actor.

[2] $primitive\_cpog(a)$ returns CPOG of a primitive actor a, $createEdge(p_1, p_2)$ creates a new edge from port p_1 to p_2.

```
Algorithm 6: $composite\_cpog(CA, M)$:Derive CPOG for Composite Actor
Input: Composite Actor $CA$, Model $M$
Output: CPOG $G = \langle V, E, X, \rho, \phi \rangle$ for $CA$
Initialize $G = \langle \{\}, \{\}, \{\}, \{\}, \{\} \rangle$;
Let $A_{NC}$ & $A_C$ be partition of actors in $CA$ into sets of Primitive
(Non-composite) and Composite actors resp. (present immediately under $CA$);
Let $I_{CA} = \{p_1, p_2, .., p_n\}$ be the inports of $CA$;
Let $O_{CA} = \{p_1, p_2, .., p_m\}$ be the outports of $CA$;
foreach composite actor $a \in A_C$ do
    //recursive call, ∪ represents composition of CPOGs
    $G \leftarrow G \cup composite\_cpog(a, M)$;
end
foreach primitive actor $a \in A_{NC}$ do
    //Section 7.2, ∪ represents composition of CPOGs
    $G \leftarrow G \cup primitive\_cpog(a)$;
end
foreach $p_i \in I_{CA} \cup O_{CA}$ do
    Let $ch_{in}$ be the in-coming channel connected to $p_i$;
    Let $p_{e_{in}}$ be source port of the channel $ch_{in}$;
    foreach out-going channel $ch_{out}$ from $p_i$ do
        Let $p_{e_{out}}$ be destination port of channel $ch_{out}$;
        Let $e_{new} = createEdge(p_{e_{in}}, p_{e_{out}})$;
        $E \leftarrow E \cup \{e_{new}\}$;
        $\phi(e_{new})$ = Constraints on $ch_{in}$ && Constraints on $ch_{out}$;
    end
end
return $G$;
```

Proposition 3 *$composite\_cpog(\mathrm{A}, M)$ (Algo. 6) outputs the CPOG representing the schedule of execution of A.*

Proof: By structural induction on the structure of the composite actor A.

3.5.2.1 Example MRICDF model and it's CPOG

Consider the example MRICDF model, and its corresponding SIGNAL code as shown in Figure 33. Despite this example being contrived, it is sufficient to communicate our idea. From the previous section, we know how to derive the sets involved in the quintuple for each actor. Composing all the actor quintuples we derive the CPOG of the whole model. The textual and the graphical representation of the CPOG is shown in Table 9 and Figure 32.

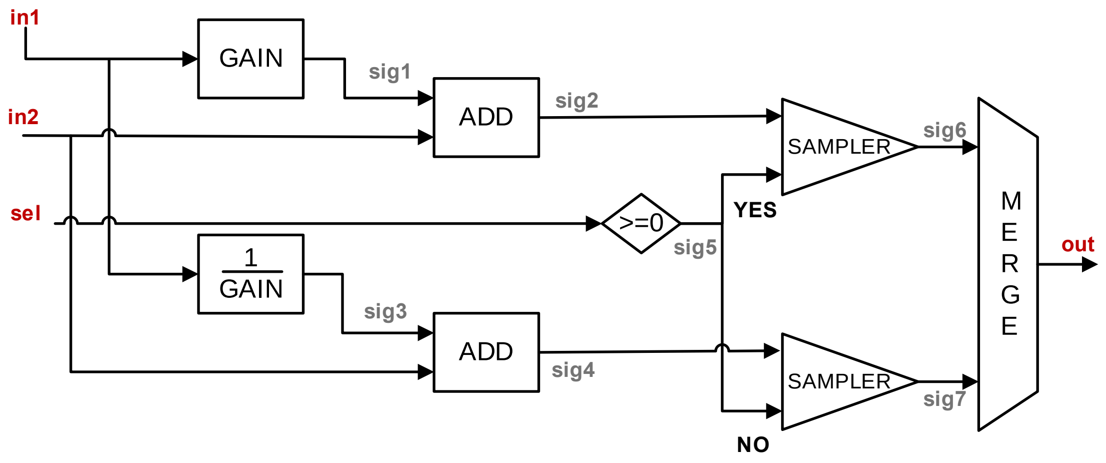


Figure 32: Sample MRICDF model

```
function process = (?int i1, i2, sel; !int out;)
(|sig1 = GAIN(in1)
  |sig3 = 1/GAIN(in1)
  |sig2 = ADD(sig1, in2)
  |sig4 = ADD(sig2, in2)
  |sig5 = (sel >= 0)
  |sig6 = sig2 when sig5
  |sig7 = sig2 when not sig5
  |out = sig6 default sig7
  |)
where
   integer sig1,sig2,sig3,sig4,sig5,sig6,sig7;
end;
```

Figure 33: SIGNAL code (ADD, Comparator, GAIN & $\frac{1}{GAIN}$ *are predefined function actors)*

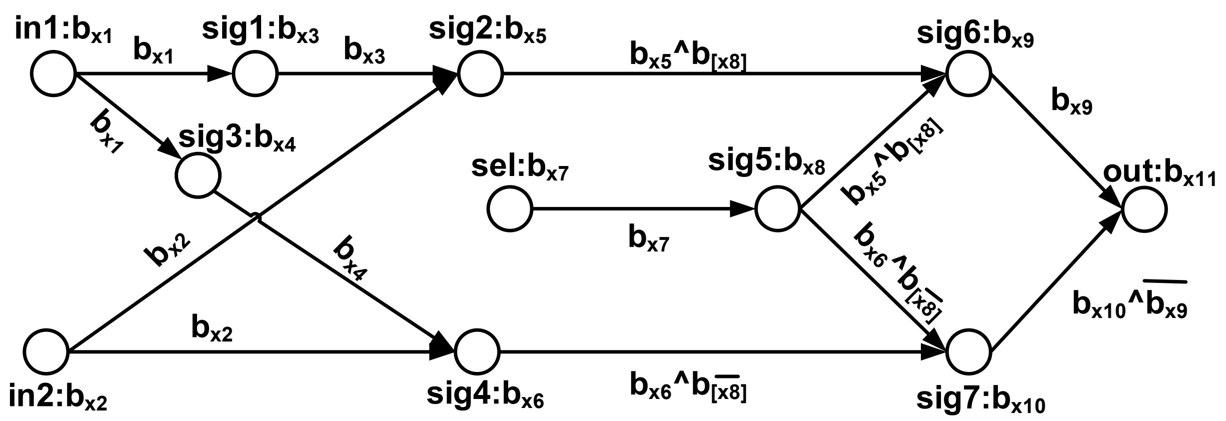


Figure 34: CPOG for the MRICDF Model

Table 9: Formal representation of CPOG for model in Figure 32

Quintuple Element	Set Elements
V	$\{in1, in2, sel, out, sig1, sig2, sig3, sig4, sig5, sig6, sig7\}$
E	$\{in1 \to sig1,\ in1 \to sig3,\ in2 \to sig2,\ in2 \to sig4,\ sig3 \to sig4,\ sig1 \to sig2,\ sig2 \to sig6,\ sig4 \to sig7,\ sel \to sig5,\ sig5 \to sig6,\ sig5 \to sig7,\ sig6 \to out,\ sig7 \to out\}$
X	$\{b_{x1}, b_{x2}, b_{x3}, b_{x4}, b_{x5}, b_{x6}, b_{x7}, b_{x8}, b_{[x8]}, b_{[\overline{x8}]}, b_{x9}, b_{x10}, b_{x11}\}$
ρ	$\{b_{x1} = b_{x2} = b_{x3} = b_{x4} = b_{x5} = b_{x6} = m,$ $b_{x7} = b_{x8} = n,$ $b_{x8} = b_{[x8]} \vee b_{[\overline{x8}]},$ $false = b_{[x8]} \wedge b_{[\overline{x8}]},$ $b_{x9} = b_{x5} \wedge b_{[x8]},$ $b_{x10} = b_{x6} \wedge b_{[\overline{x8}]},$ $b_{x11} = b_{x9} \vee b_{x10}\}$

ϕ	$\{\phi(in1)=b_{x1}, \phi(in2)=b_{x2}, \phi(sig1)=b_{x3},$ $\phi(sig2)=b_{x5}, \phi(sig3)=b_{x4}, \phi(sig4)=b_{x6},$ $\phi(sel)=b_{x7}, \phi(sig5)=b_{x8}, \phi(sig6)=b_{x9}, \phi(sig7)=b_{x10}, \phi(out)=b_{x11},$ $\phi(in1 \to sig1)=b_{x1}, \phi(in1 \to sig3)=b_{x1},$ $\phi(in2 \to sig2)=b_{x2}, \phi(in2 \to sig4)=b_{x2},$ $\phi(sig1 \to sig2)=b_{x3}, \phi(sig3 \to sig4)=b_{x4},$ $\phi(sig2 \to sig6)=b_{x5} \wedge b_{[x8]},$ $\phi(sig4 \to sig7)=b_{x6} \wedge b_{[\overline{x8}]},$ $\phi(sel \to sig5)=b_{x7}, \phi(sig5 \to sig6)=b_{x5} \wedge b_{[x8]},$ $\phi(sig5 \to sig7)=b_{x6} \wedge b_{[\overline{x8}]},$ $\phi(sig6 \to out)=b_{x9}, \phi(sig7 \to out)=b_{x10} \wedge \overline{b_{x9}}\}$

3.5.3 Transformations, Resource Estimation and Implementability

The CPOG obtained initially is first simplified before transformations are applied. The simplification step is targeted to reduce the number of variables in X depending on the equivalence relations in ρ. Also, the terms and expressions in ρ and ϕ may be updated and simplified. Algorithm 7, lists the simplification step.

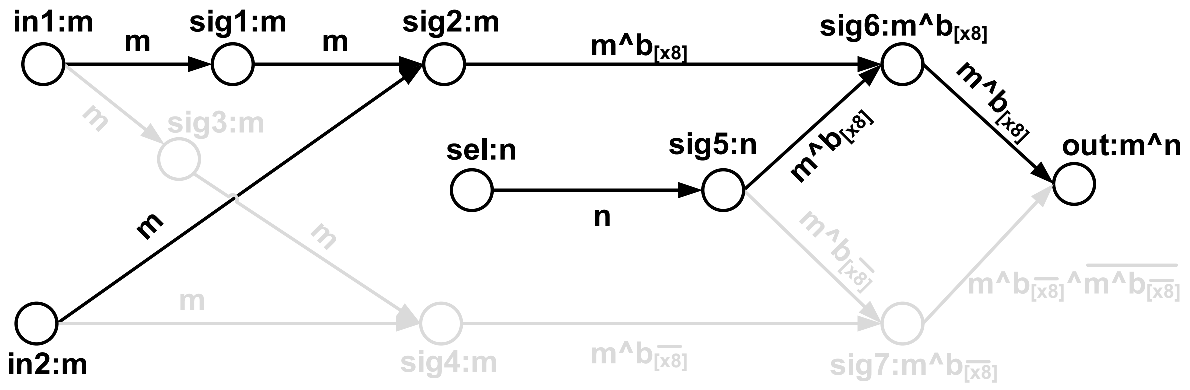
```
Algorithm 7: $simplify(G)$: Simplify CPOG
Input: Un-simplified CPOG $G=\langle V, E, X, \rho, \phi \rangle$
Output: Simplified CPOG $G=\langle V, E, X, \rho, \phi \rangle$
Let $\mathcal{E}$ = {Set of all Boolean equalities among single literals in $\rho$};
Let $\langle b_{x1}, b_{x2}, ...., b_{xn} \rangle$ represent the vector $X$;
foreach $b_{xi} \in V$ do
    if $(b_{xi} = b_{xj}) \in \mathcal{E}$ then
        replace all occurrences of $b_{xj}$ in $\rho$ and $\phi$ and simplify with idempotence and other Boolean simplification laws to
        obtain new $\rho$, and new $\phi$.
        $X = X - \{b_{xj}\}$;
    end
end
```

Proposition 4 *Algorithm 7 converges and reduces the number of control states of the resulting system.*

Proof: Convergence is based on number of equivalence classes of control variables in X, and its reduction in each step.

However, note that if we enhance this algorithm to prove more Boolean equivalences that involve numeric conditions, and values of signals based on function computation, using appropriate SMT solvers, we can reduce the number of control states further, and discover more mutually exclusive paths. So as presented, the algorithm does not necessarily reduce the number of variables in X (instruction opcodes) to the minimal possible. In general, the minimum X can be easily shown to be undecidable. However, with powerful SMT solvers of today, we can do better than just using standard variable replacement and simplifications.

In our example, ρ has the relations $b_{x1} = b_{x2} = b_{x3} = b_{x4} = b_{x5} = b_{x6}$ and $b_{x7} = b_{x8}$. Thus, the set of Boolean equalities, $\mathrm{E} = \{(b_{x1} = b_{x2}), (b_{x1} = b_{x3}), (b_{x1} = b_{x4}), \ldots\ldots, (b_{x7} = b_{x8})\}$. After simplification, the set of Boolean variables X reduces to $\langle b_{x1}, b_{x7}, b_{[x8]}, b_{[\overline{x8}]} \rangle$. For readability purposes, we have used $b_{x1} = m$ and $b_{x7} = n$. Thus $X = \langle m, n, b_{[x8]}, b_{[\overline{x8}]} \rangle$. Expressions such as $b_{x11} = b_{x9} \vee b_{x10}$ get reduced to $b_{x11} = b_{x5} \wedge b_{x8} = m \wedge n$ after propagating the changes and simplifying. By assigning $\{0,1\}$ values to each of the variables in X, one can capture all the behaviors of the system. Theoretically there can be 16 different combinations. But practically, the values assigned should not conflict with any of the ϕ functions. Thus, not all combinations of values are possible to be assigned and also of the possible assignments, some of the behaviors may be equivalent and some of them might not result in a feasible system behavior. Using a solver such as *all solution SAT solver*[31], one can find all possible assignments to the Boolean variables. Also, one has to note that if the ρ or ϕ contains constraints which involve numerical expressions, then establishing equivalence or obtaining all solutions will require advanced solvers such as SMT solvers or solvers based on Polyhedral grids, etc. We also understand that, theoretically, it might seem like there are exponentially many solutions, but for most practical applications the number of feasible solutions is much lower. We can further reduce the number of feasible behaviors, by eliminating equivalent behaviors. In our example, the assignments of the form $0xxx$ or $x0xx$ or 1111 (x being a don't care) to Boolean variables, is either not possible or leads to an impossible behavior. Only feasible behaviors of the system are obtained when the vector is of the form 1110 or 1101.



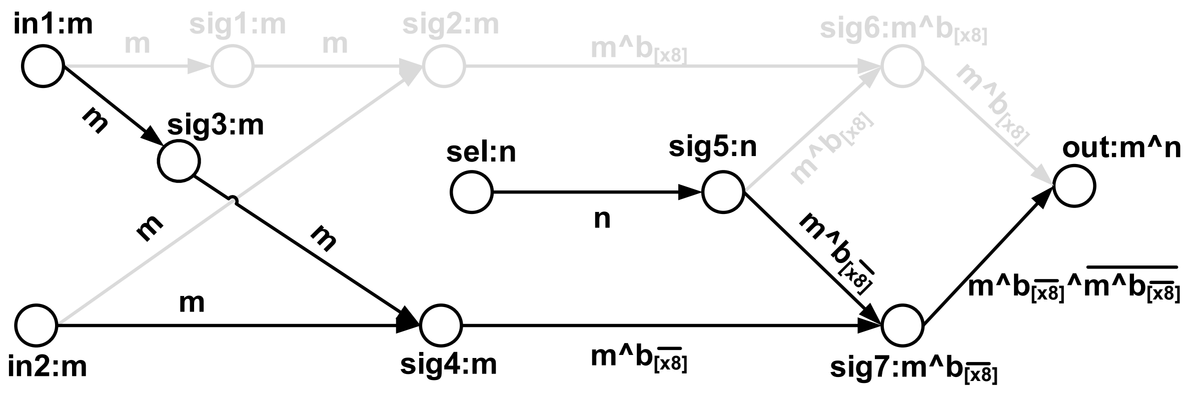


Figure 35: Modified CPOG with Boolean vector 1101 *and* 1110 *respectively* ($b_{x1} = m, b_{x7} = n$)

3.5.3.1 Resource Estimation

After determining the assignments of X for feasible behaviors, we propagate these values on to the CPOG. During this transformation of the CPOG we follow these rules,

- If the Boolean variable/expression corresponding to a node/edge has 0 as its value, then that particular node/edge is excluded from the CPOG representing current behavior as it does not contribute to it.
- If all the incoming edges to a node are excluded, then the node is also excluded.
- If all the outgoing edges of a node are excluded, then the node is also excluded.
- All edges originating from an excluded node are also excluded.
- All edges terminating on an excluded node are also excluded.
- All other nodes and edges are left as such.

```
Algorithm 8: $getFeasibleCPOGs(G,\mathcal{F})$
Input: Simplified CPOG $G$=$\langle V, E, X, \rho, \phi\rangle$ , Feasible behavior assignments for $X$ as $\mathcal{F}$={$\langle f_1\rangle$ ,..., $\langle f_k\rangle$ } //Ex:  $\mathcal{F}$={$\langle 1101\rangle$
        ,$\langle 1110\rangle$ }
Output: Set of CPOGs $\mathscr{V}$={$G_1, G_2, ..., G_k$}
Let $\mathscr{V}$ = {};
foreach feasible behavior $f_i \in \mathcal{F}$ do
    Let $G_i$ be an instance of $G$;
    foreach node or edge $z \in G_i$ do
        //Evaluate $\phi(z)$ based on $f_i$ value
        if $\phi(z)|_{f_i} == 0$ then
            $G_i = G_i - \{z\}$; //Remove $z$ from CPOG
            //Remove unused edges
            if $z$ is node then
                Remove all incoming edges to $z$;
                Remove all outgoing edges from $z$;
            end
        end
    end
    //Remove isolated nodes
    foreach remaining node $z \in G_i$ do
        Let $I_z$ be the set of incoming edges to $z$;
        Let $O_z$ be the set of outgoing edges from $z$;
        if $I_z == \{\}$ or $O_z == \{\}$ then
            $G_i = G_i - \{z\}$; //Remove node $z$ from CPOG
        end
    end
    $\mathscr{V} \leftarrow \mathscr{V} \cup G_i$; //Add $G_i$ to set $\mathscr{V}$
end
return $\mathscr{V}$;
```

Algorithm 8 presents one way to implement the rules listed above. The output of Algo. 8, is a set of the CPOGs, where each CPOG represents a feasible behavior of the system. To estimate the resources, one can simply count the number of resources needed in each CPOG and consider the maximum. The nodes and edges of the example CPOG that are active during 1110 and 1101 behaviors are shown in Figure 35 respectively. From the figure, it is clear that in any behavior of this system, at most one adder, one sampler and one GAIN block are used. If the developer had decided to implement this model/logic as hardware without this knowledge, he/she might have created two adders, gain blocks, samplers. But in actuality, the developer just needs to implement one adder, one gain block and one sampler.

3.5.3.2 Implementability

After propagating the feasible behaviors assignment values to the CPOG, if the directed graph remains *weakly connected*, in other words, replacing all of the directed edges with undirected edges must produce a connected (undirected) graph, then an implementation may be possible if there are no causal loops. Algorithm 9 lists a simple method to test for implementability.

Algorithm 9: $isImplementable(G)$

```
Input: Simplified CPOG $G=\langle V, E, X, \rho, \phi\rangle$ , Feasible behavior assignments for $X$ as $\mathcal{F}=\{\langle f_1\rangle ,...,\langle f_k\rangle \}$
Output: True if implementable, else false
Let $\mathcal{V} = getFeasibleCPOGs(G, \mathcal{F})$;
foreach $CPOG\ G_i \in \mathcal{V}$ do
    if $G_i$ has causal loops —— $G_i$ is not weakly connected then
        return $false$;
    end
end
return $True$;
```

3.5.4 Summary

In Section 3.5, we explained a new compilation scheme for the signal/MRICDF polychronous specifications based on CPOGs. We provided algorithms to derive CPOGs for given signal/MRICDF specifications. A future direction for this work could involve modifying Algorithm 3.1 using a compaction technique, that considers the equivalency between feasible behaviors. Another future work could be to explore further the aspect of sequential and concurrent implementability by applying transformations on the CPOGs.

3.6 Polyhedral Model based Causality Detection in Polychronous models

Formal software synthesis tools apply static analysis techniques to check schedulability, and actual scheduling – during which, they often resort to sound but imprecise abstractions (Ex: Boolean abstraction) which may result in a specification been rejected as non-synthesizable when it is actually synthesizable. We therefore propose to integrate recently developed decision procedures (SMT solvers) into the synthesis engine. Assuming that we successfully integrate such decision procedures inside the synthesis engine and lower the abstraction level by refinement – thus enhance the precision of the analysis – we can make more precise decisions and thereby accept a larger class of specifications for synthesizability. We can even go further as follows. Suppose a specification is rejected because it violates an invariant property, or because of cyclic dependency; and suppose that such violation is confined in a very limited area of its reachable state space. For such specifications, instead of rejecting the specification outright, the synthesis tool should guide the user by showing the exact ranges of the input values (or equational relationships between the inputs as appropriate) that could direct the resulting program to such violating area of the state space. The user may then choose an option to synthesize the program with an automatically generated wrapper. Such a wrapper will monitor the input values and when the conditions on inputs satisfy the violating condition, it could filter the inputs. Such filtering mechanism must of course be meaningful in the context of the application.

In Section 3.6, we focus particularly on a formal specification and code synthesis framework – Polychrony. It accepts polychronous dataflow specifications (Signal specifications), and compiles them to sequential C-code. During compilation, Polychrony compiler has to make various decisions regarding the specifications which are currently taken using Binary Decision Diagrams(BDDs)-based analysis. The abstraction at which the decision problems are supplied to BDDs is coarse and often leads to overcautious decisions. We can refine the abstraction and improve the decision making approach. We first show how SMT solvers with axiomatic theories could be helpful in making more accurate decisions and expand the set of acceptable specifications. We then proceed further to show how polyhedral analysis integrated into the Polychrony synthesis engine can still enhance the set of specifications that can be synthesized.

Polychrony compiler analyzes Signal specifications through a clock abstraction to check schedulability. To do this, first for each operation on signals expressed in the Signal program, a conditional data dependency relation $\rightarrow$ is derived. For example, in case of the $default$ operation, we derive $y \rightarrow x$, when y has an event, and $z \rightarrow x$, whenever y has no event but z does. We can thus draw a graph where each signal is a node, and a directed edge from a signal x to y corresponds to the above relation. However, the directed edge is valid only under certain conditions such as 'y has event' or 'y does not have event but z does' etc. This graph is called a *Dependency graph*. This graph is actually an abstraction of such a graph one could construct with E by drawing the directed edges between events based on $\rightarrow$, after all the $:$ based partition is done on E. If there is a cycle in such a graph, then one must check if the conjunction of all the conditions marking the edges in such a cycle is satisfiable. If so, there may be a cyclic dependency. If one abstracts these conditions as uninterpreted

Booleans, without modeling how they are computed, then one has a Boolean abstraction. Thus if the conjunction is found to be satisfiable, one might consider that as an indicator for existence of causal loop. However, if the computation of the conditions is traced back to the functions and predicates, then one might find that a conjunction of these conditions can never be true. In other words, the ∶ inferred over E only based on the clock calculus may be too coarse, and in reality, one may be able to refine ∶ such that events that seemed to be synchronous with respect to Boolean abstraction can now be partitioned into finer equivalence classes. Even reachability analysis, and invariants checks on such Boolean abstractions are necessarily incomplete, thus might lead to rejection of correct specifications.

Consider the example shown in Listing 3.6.1. The process ac_display has three integer input streams $minT, maxT$ and $curT$. It also has three integer output streams $disp\_coldT$, $disp\_hotT$ and $disp\_normT$. The dataflow relations constitute the body of the process description. They are separated by $|$ to indicate concurrent evaluation. During each reaction the dataflow relations are evaluated concurrently with a data dependency ordering constraint. Some of the relations are called *clock relations* which encode restrictions on relative occurrence of events on the various streams. For example, the first relation states (line 3) that the input streams are synchronous, thus all three streams would have events on them to start a reaction. This is an example of a *'clock'* constraint. The second relation states that during each reaction, the value placed on $disp\_coldT$ stream is $minT$ when $curT < 70$, otherwise it is $curT$'s value. The output stream $disp\_normT$ similarly gets either of the two values $disp\_coldT + 5$ and $disp\_hotT - 5$ based on whether $curT = 70$. The output stream $disp\_hotT$ gets either $disp\_normT + 5$ or $maxT$ depending on whether $curT > 80$. This is a contrived example of a thermostat's display process, but it is conveniently small enough to explain the various ideas described in Section 3.6.

Listing 3.6.1 Causal Loop Example

```
process AC_DISPLAY = (? integer minT, curT, maxT;
  ! integer disp_coldT, disp_hotT, disp_normT)
(| minT ^= curT ^= maxT
  | disp_coldT := minT when curT<70 default curT
  | disp_normT := (disp_coldT+5) when curT=70 default(disp_hotT-5)
  | disp_hotT    := (disp_normT+5) when curT>80 default maxT
  |);
```

A Boolean abstraction-based analysis would replace each predicate appearing in the relations by a Boolean variable taking arbitrary truth values, and will not consider the relationship between the predicates in the numerical domain of the variables in the predicate. As a result, a causal dependency loop will be detected by such analysis because of the interdependency between $disp\_normT$ and $disp\_hotT$. However, if our abstraction is cognizant of a theory of integers with ordering relations, then it would lower the Boolean abstraction to a model that considers intervals with ordering. On this model, one could prove that when $curT > 80$, only then such a causal dependency loop will exist. Obviously, if this happens, the system will behave non-deterministically or will deadlock. If this information is explicitly presented to the user upon completion of the analysis, and the user can guarantee an additional input constraint, $70 \le curT \le 80$, then generating code from this specification is legitimate – as the program will not display any deadlock behavior. In addition, if one wants to ensure safety, one could produce a wrapper that would intercept all inputs $curT$ and check against this constraint, and filter out any occurrence of input value that violates the user guaranteed constraints. However, if the user can guarantee only $70 \le curT \le 90$ – the system will exhibit causal behavior when $80 < curT \le 90$. But the system has a safe operating area, $70 \le curT \le 80$. One could still apply a wrapper to prevent the system from moving outside its safe operating area – if it makes sense for the application. Our proposed solution approach is described below.

Solution approach

Given a polychronous specification, if a Boolean abstraction-based analysis finds the specification to violate certain safety properties, automatically refine the abstraction level to

1. consider a theory of integers or rationals with additions and multiplications by constants
2. verify(either using SMT method or Polyhedra method) if such abstraction lowering still finds the same violation
3. if not apply the synthesis step
4. if violation is still present – it may be due to our inability to handle theories beyond what we considered – and hence transform the unsafe operational range in the form of input range or relationships; and provide the user with an option to generate a wrapper-based implementation that filters out certain inputs to keep the program in safe operational trajectories.

This will enhance the set of polychronous specifications that can be gainfully used to generate useful implementations.

3.6.1 Analysis of Polychronous Specifications

As explained in Section 3.2, in the Polychronous model, the primitive notion is that of events, which are infinite and partially ordered. These events are modeled as synchronous when they happen within one logical instant. So $\tau = \varepsilon/\sim$is the set of all logical instants, and with slight abuse of notation, we use ≤ as the partial order on the set of logical instants as well. If $t \in T$ denotes a logical instant, and $e_{1t}, e_{2t} \ldots e_{it}, \ldots \in t$, denote the events in t , the data dependency implied by the operators defines a causal order among the events. Note that no two events in t can belong to the same signal because events belonging to the same signal are strictly ordered. Thus at any instant t , the causal order among events is also the causal order between signals the events belong to. Of course, this causal order may change from one logical instant to another as the data dependencies are conditional as explained in the previous section.

In a polychronous specification (Signal program) intended for sequential implementation, we should be able to prove that ≤ over T is actually a total order [43]. Furthermore, we should be able to establish a relationship between all signals in terms of causal dependence, which is acyclic and tree structured, for schedulability. If the specification is meant for multi-threaded implementation, then ≤ over T can still be a partial order with certain properties (e.g. weak hierarchic [76]). Therefore, given a polychronous specification, one wants to statically analyze the following kinds of questions:

(i) Is the ≤ over T a total order?

(ii) Is ≤ over T a partial order but with weak hierarchic property (a la [76])?

(iii) Is there any logical instant where the causal dependence is cyclic? (i.e., is there a causal loop or causality problem in the specification?) (i.e existence of causality cycle)?

(iv) Is there any logical instant which is unreachable given a condition on inputs?

(v) Is there any logical instant where an invariance between signals is violated?

Methods for answering questions (i), and (ii) have been considered elsewhere (c.f. [43], [76]). In [40], one method for answering question (iii) for polychronous specifications using SMT solvers has been addressed. In Section 3.6, methods for addressing (iv) and (v) with an aim to salvage the specification to be useful on certain input ranges are considered. Usually, these questions can be answered by using abstract interpretation of the Signal program in a Boolean domain with the possibility of imprecision. For instance, cyclic dependence may be dependent on predicates on signals assuming a particular value or a set of values – such as "$x > 10 \wedge y < 5$". In a Boolean abstraction satisfiability of these where $x > 10$ and $y < 5$ are independent propositions is guaranteed, but, if we consider the computation of x and y and use a theory of integers to model that, we might conclude that this condition can never be true. SMT solvers are apt in answering such questions [40], provided the theory we require are decidable or at least semi-decidable.

Consider the Signal program shown in Listing 3.6.2, which is an extension of the program shown in Listing 3.6.1. When a Boolean abstraction is analyzed, it identifies the possibility of causal loop because of the interdependency between $disp\_hotT$ and $disp\_normalT$ as shown in Listing 3.6.3.

Listing 3.6.2 True Causal Loop

```
process AC_DISPLAY = (?integer minT, curT, maxT, curP, curK;
                      !integer disp_coldT, disp_hotT, disp_normalT)
(| minT ^= curT ^= maxT ^= curP ^= curK
% Conditions %
  | cond_1 := ((curT >= 2) and (curT <= 18))
  | cond_2 := ((curP >= 3) and (curP <= 21))
  | cond_3 := ((curK >= 25) and (curK <= 35))
  | cond_4 := (curT-curP >= -10)
  | cond_5 := ((curT+curP >= 11) and (curT+curP <= 33))
% Output Computation %
  | disp_coldT    := minT when (curT<minT) default curT
  | disp_normalT:= (disp_coldT+10) when
                         (not(cond_1 and cond_2 and cond_3))
                         default (disp_hotT-10)
  | disp_hotT     := (disp_normalT+10) when(cond_4 and cond_5)
                         default maxT
  |)
where
  boolean cond_1, cond_2, cond_3, cond_4, cond_5;
end;
```

Listing 3.6.3 Possible Causal Loop

```
(| {disp_hotT --> disp_normalT} when C_CLK_31
   | {disp_normalT --> disp_hotT} when C_CLK_23
   |)

   where, C_CLK_31 = cond_4 and cond_5
          C_CLK_23 = cond_1 and cond_2 and cond_3
```

One can invoke an SMT solver to check for nullity of clock constraints ($C\_CLK\_31 \wedge C\_CLK\_23$) on the path of the apparent loop. This SMT extension to Polychrony is explained in next section.

3.6.2 SMT extension to Polychrony

In the earlier section it was shown that the decisions taken by abstracting *SIGNAL* specifications to Boolean logic can be overcautious leading to rejection of correct specifications. We lowered the abstraction and augmented the Polychrony tool-set by adding YICES SMT solver extension to make more precise decisions. The execution flow of the augmented polychrony compiler is shown in Figure 37. The compiler first parses the *SIGNAL* specifications and constructs an abstract syntax tree(AST). From the AST, it extracts information regarding computations and creates implicit dependency subgraphs and user defined dependency subgraphs. Using these subgraphs it constructs a dependency graph. This may contain dependency loops. Causality analysis is performed as graph transformations. If there exists no causal loops, the code generation step is performed. But if the compiler reports existence of a causal loop (Listing 3.6.2), we extract the clock dependencies of the probable cycle and transform them to SMT equations. This is done by extracting the clock constraints and generating the predicates for Yices SMT solver as shown in Listing 3.6.4. Invoking Yices solver will decide this condition as *satisfiable* (which indicates the existence of true causal loops) and it outputs just *one* counter example to show a case where causal loop may create a deadlock. However, there might exist a range of input values where this deadlock could happen which SMT solver will not provide directly. Similarly, SMT solver will not provide range of input values where the causal behavior is not exhibited, in other words the bounds for safe operating region. In the next section, we present a polyhedra model-based analysis approach implemented in the Polychrony compiler to eliminate this limitation of SMT solvers.

Listing 3.6.4 Assertion in SMT solver and Solution

```
(define curT::int)
(define curP::int)
(define curK::int)

(assert (and (<= curT 18) (<= curP 21) (<= curK 35)
(>= curT 2) (>= curP 3) (>= curK 25) (<= (+ curT curP) 33)
(>= (- curT curP) -10)    (>= (+ curT curP) 11) ) )
(check)
```

Based on the output of SMT solver, we conclude if it is a true or false causal loop. If it is a false causal loop, it is possible to generate code by adding assertions and doing modular code generation. If it is a true causal loop, we raise an error and based on SMT output, we also provide valuable feedback on when the dependency loop was triggered. This approach expands the subset of the acceptable *SIGNAL* programs by the polychrony compiler with a negligible increase in total compilation time.

3.6.3 Polyhedral Model based Analysis

The polyhedral model provides a powerful abstraction domain for various static analysis techniques. A polyhedron is basically, a locus of the solutions of a system of affine inequalities and equations. Various algebraic, arithmetic and set operations can be done on these polyhedra. In the previous section we showed with an example, the limitations of SMT solver-based causality analysis technique. If the clock constrains (Listing 3.6.3) are linear expressions with arithmetic, logical and relational operators they can be translated into a system of affine inequalities and equations, which can then be analyzed using polyhedral libraries.

3.6.3.1 Constraint Extraction and Transformation

Consider the input constraints shown in column 1 of Table 10 for the Signal program shown in Listing 3.6.2. The clock constraints for possible causal loop (shown in Listing 3.6.3) are first obtained from Polychrony compiler. These constraints are parsed and automatically transformed to a system of affine inequalities and equations as shown in column 2 of Table 10. There exists an implicit logical intersection among all the constraints within each column of Table 10. The constraints in Table 10 need to be transformed into affine forms to use the *PolyLib* library [65]. This system is further abstracted to matrices before using Polylib APIs. Figure 36 shows the plot of polyhedrons representing both input constraints and true causal loop constraints. From multiple views we see that there exists a region of intersection between the two polyhedrons, which indicates the existence of true causal loops with the current input constraints.

Table 10: Input and True Causal Loop constraints

Input Constraints	Loop Constraints
10 ≤ curT ≤ 40	2 ≤ curT ≤ 18
10 ≤ curP ≤ 40	3 ≤ curP ≤ 21
10 ≤ curK ≤ 40	25 ≤ curK ≤ 35
	curT - curP ≥ -10
	11 ≤ curT + curP ≤ 33

3.6.3.2 Polyhedral Analysis

To obtain the bounds of safe operating region and the region where true causal loop exists, we apply two polyhedral operations from the *PolyLib* library. Let I be the polyhedron constructed considering input constraints and L be the polyhedron for the domain of the potential causal loop.

1. $DomainIntersection(I, L)$: This operation returns the intersection of two polyhedral domains. This is used to compute $I \cap L$.
2. $DomainDifference(I, L)$: This operation returns a new polyhedral domain which is the difference, $I - L$.

Figure 36 also shows the plots for both $I \cap L$ and $I - L$ respectively. $I \cap L$ gives the input space domain in which causal behavior is exhibited. If $I \cap L$ is empty, then the potential causal loop is not a true causal loop. $I - L$ gives the domain of safe operating area.

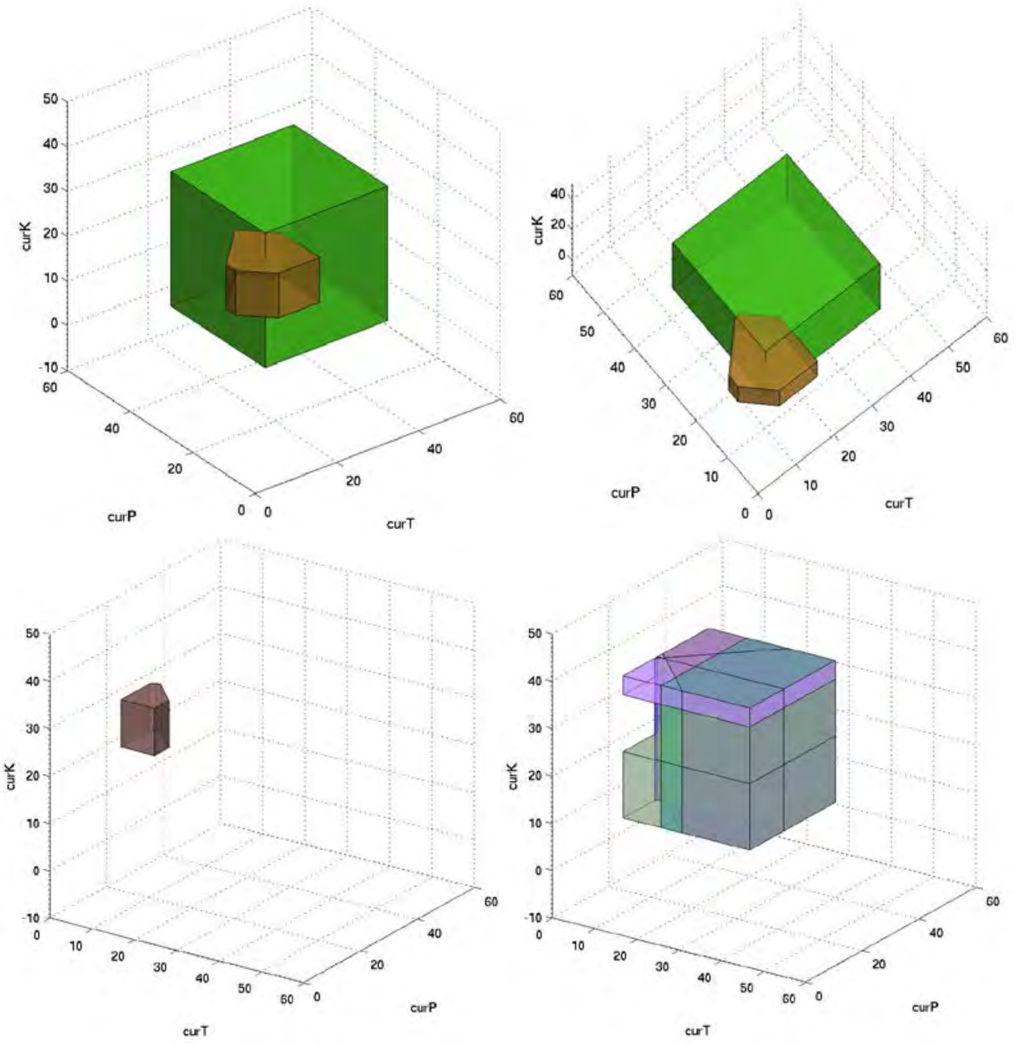


Figure 36: (Top) 3D-plot (multiple views) of Polyhedrons representing Input and Loop Constraints. $(Bottom)$ 3D plots of $I \cap L$ and $I - L$

3.6.3.3 Limitation of Polyhedral libraries

Almost all of the existing polyhedral libraries including the one we are using, $PolyLib$, have restrictions that they can only accept integer constraints. In our technique, all rational constraints are multiplied by least common multiple to obtain integers, and floating point numbers are truncated based on precision specified by the user. Then we multiply the truncated floating point constraint by a suitable number such that it becomes an integer. The truncations preserve the soundness of the technique by over-approximating the polyhedron.

3.6.3.4 Safe code synthesis using Wrapper

From the result of polyhedral analysis, we obtain the bounds on inputs, such that for any input within the bounds the system will be in safe operating region. To ensure safe operation always, we first assign the current value of the input signals to temporary variables. Then we check if the input values are within the bounds. If yes, we use the current input values stored in temporary variables. If no, we reassign the temporary variables with default values that are known to keep the system within safe operating region. The assignment of default values to temporary variables is done keeping in the mind input clocks. This wrapper code prevents any inputs violating the conditions of safety from being passed. The user of the synthesis tool is given an option to choose if such implementation makes sense in the application domain. In Listing 3.6.5[3] we show the wrapped code for the SIGNAL program shown in Listing 3.6.1.

Listing 3.6.5 Signal program of Listing 3.6.1 with wrappers

```
process AC_DISPLAY = (? integer minT, curT, maxT;
  ! integer disp_coldT, disp_hotT, disp_normT)
(|curT ^= cond_1 ^= tempCurT
  |cond_1       := ((curT >= 70) and (curT <= 80))
  |tempCurT     := curT when cond_1 default DEFAULT_VALUE

  |disp_coldT:= minT when tempCurT<70 default tempCurT
  |disp_normT:= (disp_coldT+5) when tempCurT=70 default
                        (disp_hotT-5)
  |disp_hotT := (disp_normT+5) when tempCurT>80 default
                        maxT
  |)
  where
       bool cond_1;
       integer tempCurT;
  end;
```

3.6.3.5 Implementation and Design Flow

We have enhanced the open source Polychrony compiler obtained from [81] by integrating the Yices SMT solver [86] and *Polylib*. In particular, as of current implementation, we apply only causal loop detection, and provide the corresponding wrapper generation. The execution flow of the enhanced polychrony compiler is shown in Figure 37. The input Signal specifications are parsed by the compiler and an abstract syntax tree (AST) is created. Transformations are applied on AST to get a directed graph. Causality analysis is done on the graph and possible causal loops are listed by the compiler. There are two different approaches to identify the true or false causal loops. If no input constraints or no safe operating bounds are requested, we can use the SMT-based technique to identify the true or false causal loops. If bounds on safe operating region are requested, then we parse the input and loop constraints, generate polyhedra models and invoke *PolyLib* for the analysis. Based on the SMT output or result of polyhedral analysis, we conclude if it is a true or false causal loop. We proceed synthesizing the code for safe operating region using the technique described earlier.

[3] DEFAULT_VALUE is a value that when assigned to input signals, is known to keep the system in safe operating area.

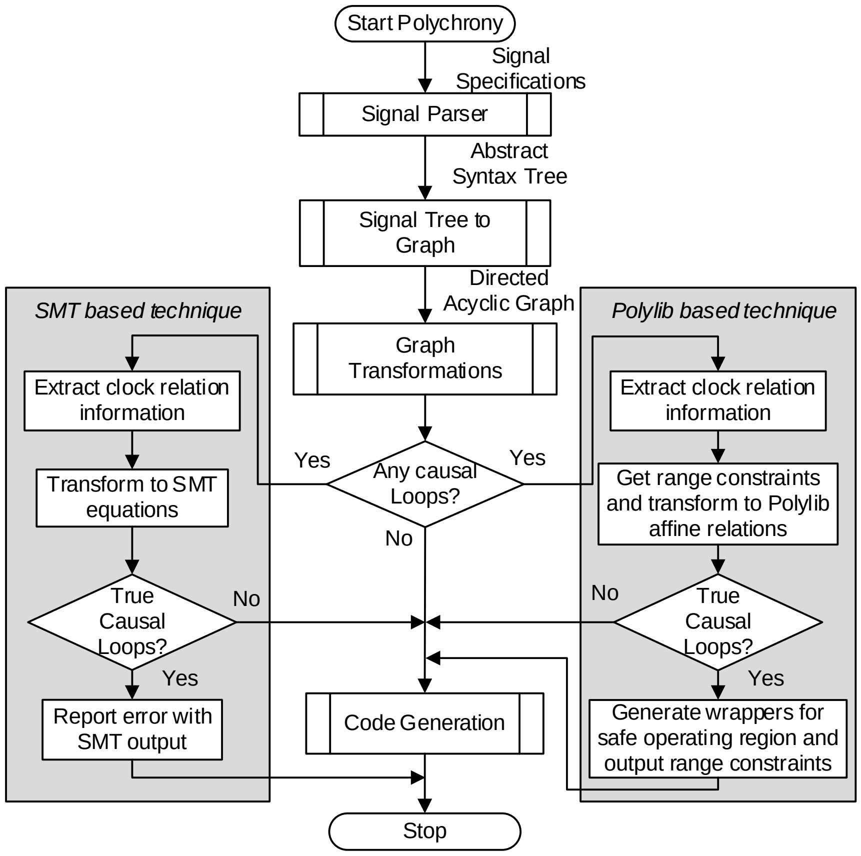


Figure 37: Execution flow of enhanced polychrony compiler

3.6.4 Summary

In Section 3.6, we showed how to augment the Polychrony compiler with YICES SMT solver for making better decisions and illustrated it with a SMT-based causality analysis technique. The proposed SMT based solution adds minimal overhead to the compilation time, and it can be easily proven to be sound. We also presented an integration of polyhedral analysis to existing static analysis techniques for polychronous specifications to obtain safe operating ranges for all inputs. By doing so we enhanced the existing static analysis techniques and expanded the subset of Signal specifications that the Polychrony compiler can accept, with minimal overhead addition to the compilation time. The proposed technique does not account for dynamic behavior of variables. The current polyhedral library we use, *Polylib* [65], is restricted to integer and approximate floating point constraints expressed as linear system of inequalities and equations. This is a restriction on the library and not on the technique we proposed. In future we plan to use a different library which can handle floating point constraints. We also want to expand the analysis beyond polyhedra into non-linear system of inequalities and equations. We also plan in future to enhance the compiler to do static analysis for checking other properties beyond causality.

3.7 Type Inference and Type Consistency Verification of Polychronous models

Among the leading causes for failure of embedded software, a prominent one is mismatched assumptions about signals at the interfaces of various components. In a model-driven design methodology, such mistakes happen at the modeling time. In [61], a number of cases have been cited, many of which show that when two software components are integrated, inconsistencies in the assumptions with respect to dimensions and units of signals at the interfaces, could lead to failure of the entire system. Traditional type checking done by compilers after the software is synthesized or coded from the model can only ensure consistency between the traditional data types (float, int, bool, double etc.,) of the signals at the interface. But, this analysis totally ignores the dimensional and unit inconsistencies. As a result of this, signals with different dimensions (e.g., velocity and acceleration) or signals with same dimensions but with different units (e.g.,velocity in m/sec and km/hr) could mistakenly be connected. In [49] and [50], this problem has been formulated as an ontology issue, and ontology aware extensions of the Ptolemy signal types and corresponding checks have been implemented. In [67], a similar extension has been proposed for Simulink models. In both of these, the model of time was not polychronous, making it less complex to extend the type system.

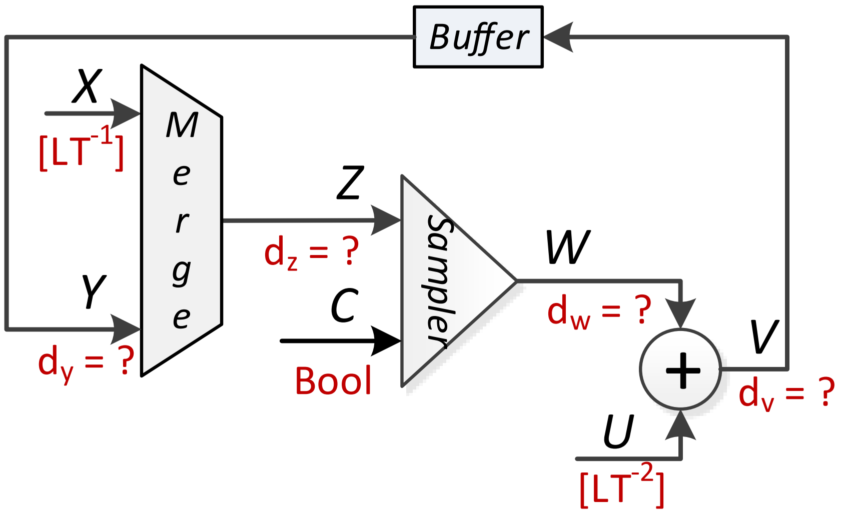
To find the bugs attributed to the mismatch in dimensions and units at the interfaces of composed components, there could be two approaches:

(i) extend the type system of the modeling language; or
(ii) extend the type system of the target software language.

However, the second approach requires a change in a standard language such as C/C++/Java etc., and a change in their compilers, which requires considerable effort and compatibility with standards. On the other hand, if a run-time dynamic checking is implemented, then it results in increased run-time overhead. Therefore, if a formal model-driven correct-by-construction approach is followed, it is more logical to embed the type information in the models and statically check for type consistency at the interfaces by extended type checking algorithms. Since in the model-driven approach, the source of such errors are in the model itself, early detection of such errors can improve the quality of models – hence the quality of the synthesized software.

Many modeling languages including Simulink have a synchronous model of time, in the sense that, signals at interfaces can all be read if present, and sensed if absent (usually absence is coded in terms of default or unchanged values). Thus for such languages, the extension of a type system with dimensions and units poses less of a problem than in polychronous modeling languages such as Signal[9]/MRICDF[43]. The polychronous timing model poses a few additional challenges:

(i) the interface signals are not all synchronous – thus, a signal may be absent or present during a particular reaction;
(ii) a modeling construct to merge multiple signals creates union data types; and
(iii) the same signal may carry data of different dimensions and units during different reactions due to the merge construct – hence a clock calculus must be part of the type checking extension to handle such *polychronous or union types*.



```
function SignalProgram =
(? real X, U; ? boolean C;
 ! real V;)
(|Z := X default Y
 |W := Z when C
 |V := W+ U
 |Y := V $ init 0.0
 |)
  where
     real Z, Y, U, W;
  end;
```

Figure 38: Example MRICDF model

Consider the MRICDF model shown in Figure 38. Despite the model being small and contrived, it suffices to illustrate the problem. The model has three inputs - X, C & U, one output V, which is delayed and fed back as input Y. Signal Z is resultant of priority merging of signals X and Y, with priority given to first input (X). Further, signal Z is sampled using signal C to yield signal W. All the signals are of *double* data-type. Signal X has dimensions[4] of $[LT^{-1}]$, while C is an adimensional signal and U has $[LT^{-2}]$ as dimensions. The dimensions of rest of the signals are unknown and denoted by $d_{<sig\_name>}$. For this model to be type consistent, the following constraints have to be satisfied.

- As signals have to be of the same dimension to be added: $d_v = d_w = d_y = [LT^{-2}]$
- By definition of sampler actor(explained in Section 1): $d_w = (d_z$ when $(C = true))$
- For consistency: Signal X is absent when $C = true$
- Further inference: $d_y = (d_z$ when $(C = true))$

If we know that, signal X is present when $C = true$, then $d_z = [LT^{-1}] = d_w$, which results in inconsistent typed signals W and U being added together, breaking the type consistency of the model. If we do not consider the clocks of signals, then the type of signal Z is of type X or Y, i.e, a union data type. Thus, signals in such systems may have tagged union types, tagged with clock information associated with signals. To ensure type correctness of such systems, we need to match the dimensions and units of the signals along with their data types by considering the clocks of the signals.

[4] We are assuming the reader is familiar with the way units and dimensions are expressed as per SI Standard.

We therefore propose a polychronous type system extension for the MRICDF language and extend the EmCodeSyn tool with a framework that allows users to specify unit and dimensional information for some/all signals. We also define type inference rules, which an algorithm uses to infer unit & dimensional information for the rest of the signals by considering the clocks of the signals. After the inference algorithm assigns types for all signals, a correctness checking algorithm verifies type consistency of the model with the help of an SMT solver and if it can verify completely, it provides a set of clock constraints associated with the unit & dimensional information under which the model is guaranteed to be type consistent. If it can't verify, it provides a set of constraints that causes type inconsistency in the model.

Our major contributions described in Section 3.7 are:

- For polychronous languages, for the first time, we proposed a polychronous type system with tagged union types containing clock information and implement this type system in EmCodeSyn framework. Our framework also allows users to specify application specific type information to store dimensional and unit information of signals during modeling.
- We proposed type inference rules and an inference algorithm that folds in clock calculus – for polychronous modeling languages. Even though the implementation is done in the context of MRICDF, it applies to Signal language as well.
- Further, we proposed a fully automated SMT-based verification approach that checks for type consistency and enables the framework for early detection of modeling bugs associated with dimensions/units/clocks for interface signals.

Even though, for Simulink and Ptolemy, type extensions have been implemented, we believe that the polychronous model of time added additional complexity to make our type system novel and the type inference & consistency checking approach completely distinct – especially with the necessity of clock calculus in the type checking process.

3.7.1 Type System and Inference Rules

As mentioned in Section 3.2, an MRICDF model is a composition of primitive and composite actors, where actors are connected using channels. These channels represent the physical signals that carry values corresponding to one or multiple physical quantities out of the infinitely many physical quantities. Keeping practicality in mind, we have considered an exhaustive set of physical quantities, which enables us to model most of the physical systems. We classify the dimensions of physical quantities into three categories (a) *Fundamental dimensions* - Ex: Mass, Length, Time etc, (b) *Derived dimensions* - Ex: Momentum, Velocity etc, and (c) *Union dimensions* - Ex: Momentum/Velocity, Pressure/Force etc. Fundamental dimensions represent dimensions for a set of physical quantities from which we can derive dimensions for other physical quantities. Derived dimensions represent the dimensions of physical quantities which are derived from fundamental dimensions. Union dimensions are the dimensions that represent merged signals that have a combination of either non-union dimensions or union dimensions or a combination of both. They represent the dimensions of signals that are of the union data type. In real life, union typed signals arise when data representing multiple physical quantities are time multiplexed and sent over same physical signal. At any given instant, the union typed signal can only take one of the multiple dimensions possible. Our type system contains units and dimensions corresponding to each of the physical quantity in the exhaustive set of physical quantities. Along with the types representing physical quantities, we also have special types denoted by $Bottom(?)$, $Top(\mathrm{T})$ and $Absent(\perp)$, which represent an unknown data type, an inconsistent data type and absent type respectively. Along with each dimension, we also store the corresponding unit information in SI, CGS, MKS and/or user defined format including the multiplication factor.

3.7.1.1 Type Set

Let M be any MRICDF specification and S be the set of all signals in M. Let D be the set of all possible dimensions which can be assigned to signals in S and let U be the set of all possible units for each of the dimensions in D. Also, let C be the set of all the clocks of M. We represent a non-union type of any signal as a tuple $\langle d,u,\hat{c}\rangle$ and a union type of any signal as a tuple $\langle d_1/d_2/..,u_1/u_2/..,\hat{c}_1/\hat{c}_2/..\rangle$, where $d,d_1,d_2 \in D$, $u,u_1,u_2 \in U$ and $\hat{c},\hat{c}_1,\hat{c}_2 \in C$. Let B be the set of all such possible tuples.

$$B = \{\langle d,u,\hat{c}\rangle,...,\langle d_1/d_2,u_1/u_2,\hat{c}_1/\hat{c}_2\rangle,..\}$$

$$\forall\, d,d_1,d_2 \in D, u,u_1,u_2 \in U, \hat{c},c_1,c_2 \in C$$

B represents the Type set.

When we say a signal s is of non-union type $\tau = \langle d,u,\hat{c}\rangle$, it means the dimension and unit of the physical quantity whose value flows through signal s is d and u respectively, at clock $\hat{c}$. Similarly if signal s is of union type $t_1/t_2 = \{\langle d_1/d_2,u_1/u_2,\hat{c}_1/\hat{c}_2\rangle\}$, then we say that the dimension and unit of signal s is d_1 and u_1 respectively at clock $\hat{c}_1$ and it's dimension and unit at clock $\hat{c}_2$ is d_2 and u_2 respectively. An example of a tuple for a non-union typed signal sig_1 is $\langle time, sec, \hat{x}\rangle$ and an example tuple for a union typed signal sig_2 is $\langle time/temperature, sec/Kelvin, \hat{x}/(\hat{y}-\hat{x})\rangle$.

In the union type $t_1/t_2 = \{\langle d_1/d_2,u_1/u_2,\hat{x}/\hat{y}\rangle\}$, d_1/d_2 is actually $d_1 \grave{o} d_2$, where $\grave{o}$ is a union operator for the data types. We also define a tagged union variant where $t_1/t_2 @ \hat{x}$ refers to the union type manifested at clock x by the union type. If after the clock calculus, it turns out that t_1/t_2 would manifest at clock x as non-union type t_1 then $t_1/t_2 @ \hat{x}$ = t_1. If the clock calculus indicates that at clock x, the union type will manifest as t_2, then $t_1/t_2 @ \hat{x}$ = t_2, else we don't have enough information to resolve $t_1/t_2 @ \hat{x}$. Based on these definitions, we now describe the inference rules that can be used to infer the types of each signal. Inferring types for the model M is done by repeatedly applying the rules until all the signals are assigned with some type or we reach a conflicting assignment.

Let x, y and z be any three signals, τ, τ', t_1 and t_2 be any four types and let Γ be the type environment containing the type assignments such as $x:\tau$.

3.7.1.1.1 Buffer Actor

Operation: y = buffer(x)

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash y : \tau} \qquad \frac{y : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

The type inference rule states that, if "x is present and is of type τ" is in the type environment Γ, then we can infer that "y is also present and is of type τ" in the same type environment Γ and vice versa. In other words, if either the type of input or output signal of the buffer actor is known and the type of other one is not known, then the data type of the unknown signal is same as that of the known signal. For readability purposes, we are not showing the trivial clock details of the signals (Ex: $\hat{x} = true$, $\hat{x} = \hat{y}$, etc.) in the rules.

3.7.1.1.2 Sampler Actor

Operation: y = sampler(x, z)

$$\frac{x : \tau \in \Gamma, z : Boolean \in \Gamma}{\Gamma \vdash y : \tau @[z]}$$

If the type of the first input of the sampler actor is known, then the type of output of sampler actor is same as that of the first input – when it is present and the second input has a $true$ value. The type of output signal can be further refined by analyzing the clocks of both the input signals. A few examples of refinement are provided below.

- From the clock relation graph, if we know that $[\hat{z}] \subseteq \hat{x}$, i.e, whenever $z = true$, signal x is present, then we can say that $\Gamma \vdash y : \tau$
- If we know that $[\hat{z}] \wedge \hat{x} = \phi$, i.e, the intersection of the set of instants when $z = true$ and when signal x is present is the null set(ϕ), then we can say that $\Gamma \vdash y : \perp$
- If τ is a union type such as $\tau = t_1/t_2$ and if we know that at clock $[\hat{z}]$, the union type manifests as $t_1/t_2 @[\hat{z}] = t_1$, then we can say that $\Gamma \vdash y : t_1$
- Similarly, if the union type manifests as $t_1/t_2 @[\hat{z}] = t_2$, then we can say that $\Gamma \vdash y : t_2$

3.7.1.1.3 Merge Actor

Operation: y = merge(x, z)

$$\frac{x:\tau \in \Gamma,\ z:\tau \in \Gamma}{\Gamma \vdash y:\tau}$$

$$\frac{x:\tau \in \Gamma,\ z:\tau' \in \Gamma}{\Gamma \vdash y:\tau @\ x / \tau' @(\ z -\ x)}$$

The rules for merge actor state that, if both the input signals are of the same type, then the output signal is also of the same type. If the input signals are of different types, then the output signal is of union data type. Similar to the case of sampler actor, we can refine the type of the output signal of merge actor, by analyzing the clocks of input signals.

An example of the refinement is shown below. If the set of instants where the second input of merge is present is a subset of the set of instants where first input of merge is present, in other words, if clock of signal z is a subset of clock of signal x ($\hat{z} \subseteq \hat{x}$), then we can say that the whenever signal x is absent, signal z is also absent. Hence, we can say that type of output signal y is the same as that of the type of first input x.

$$\frac{x:\tau \in \Gamma,\ z:\tau' \in \Gamma\ and\ \ z \subseteq\ x}{\Gamma \vdash y:\tau}$$

3.7.1.1.4 Function Actor

Operation: y = f(x, z) = $x \times z$

The type of the output signal of a function actor depends upon the operation performed inside function actor. For most of the generic operations such as multiplication, integration, etc., the tool can infer the types automatically. If the function actor is doing an user defined operation, to infer its output type, we require the modelers to provide the equivalent SMT formula for the generic operation. A provision for this is made in the tool. Here we illustrate type inference for output of a function actor doing simple multiplication operation.

$$\frac{x:t_1 \in \Gamma,\ z:t_2 \in \Gamma\ and\ \ f:t_1 \times t_2 \to \tau}{\Gamma \vdash y:\tau}$$

3.7.1.1.5 Composite Actor

A Composite actor is a hierarchical composition of the primitive actors and other composite actors. We recursively explore inside the composite actor in a depth first manner, until we reach a point where there are only primitive actors, and use the rules described above to infer their types. We then propagate the types at interface hierarchically and finally infer the input and output signals of the composite actor.

3.7.1.2 Inference Algorithm

Figure 39 shows the flowchart for Inference algorithm. The algorithm basically applies inference rules to each actor in the model until all the signals are assigned with some type or a conflict in type assignment occurs.

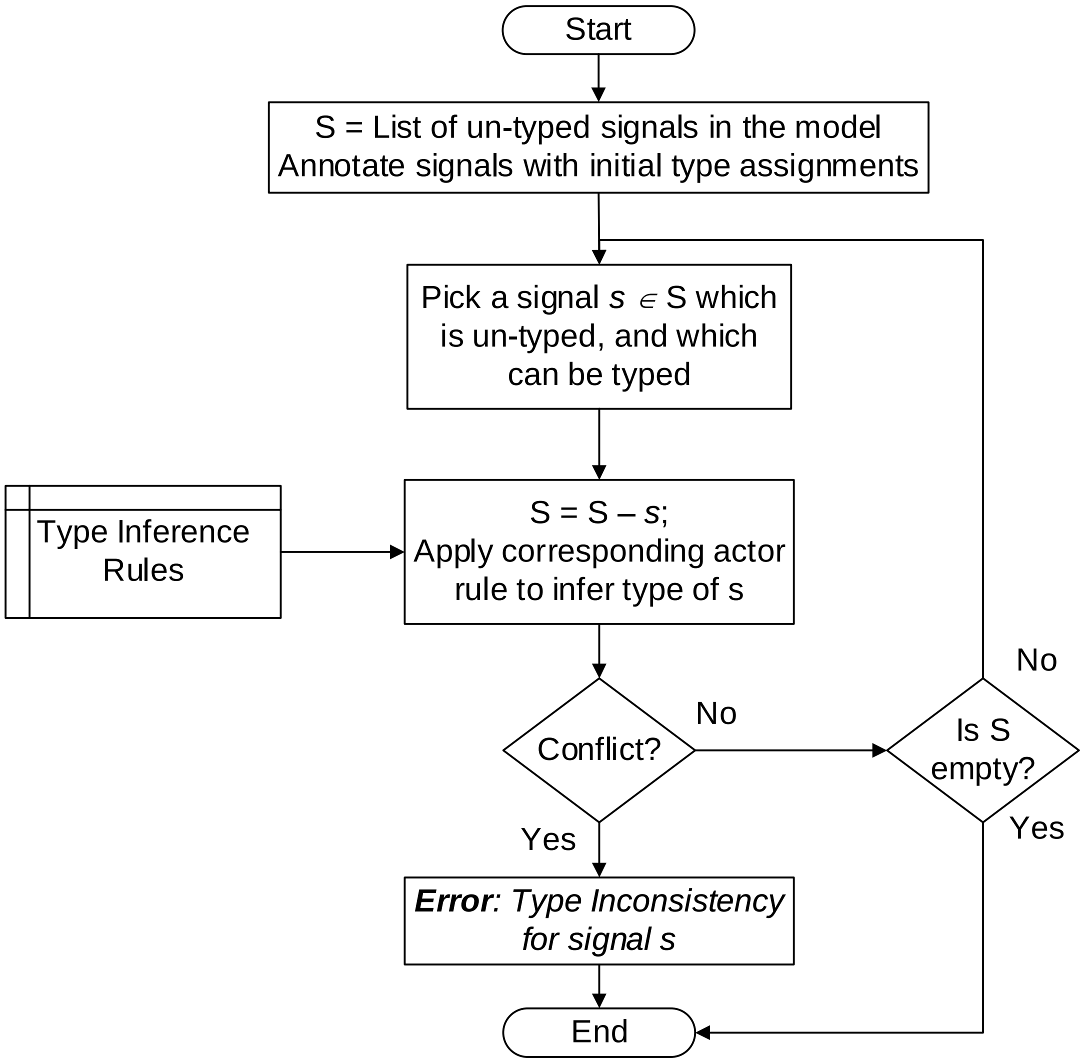

Start
S = List of un-typed signals in the model
Annotate signals with initial type assignments
Pick a signal s ∈ S which is un-typed, and which can be typed
Type Inference Rules
S = S – s;
Apply corresponding actor rule to infer type of s
Conflict?
No
Yes
Is S empty?
No
Yes
Error: Type Inconsistency for signal s
End

Figure 39: Inference Algorithm

Initially, we set the types of all signals which are to be determined as $Unknown$. Then user has to initialize types for some/all of the input signals and optionally other signals (Ex: output signals of function actor, etc). After inputting the initial types, the user can ask the tool to infer the types of the rest of the signals. The tool then starts inferring based on the inference rules for each actor described earlier until all the signals are assigned or a conflict occurs.

After the types for all the signals have been identified, one has to verify that the model is type consistent or not. Verifying type consistency is trivial when the model does not contain any signals that are of union type. In such models, we just have to verify that the connected ports have the same dimensions and units. In fact, for such models, the inference algorithm itself serves as consistency check too. But, if there are signals that are assigned with an union type, then, we also need to know if the type consistency is still upheld, when the union type manifest to non-union type under various clock constraints. This consistency check is done using SMT solver as explained in the next section.

3.7.2 SMT based Type Consistency Checking

A model is said to be always type consistent when we can ensure type consistency in all possible instants. At any given instant, any union typed signal in the model has to be resolved to a unique non-union type, which is one of the constituent types. To do so, we have to resolve the union typed signals under various clock constraints and see to what non-union type do they manifest to under those constraints. This can be done by using the clock relations in the hierarchical clock relation graph. After resolving we check if the type consistency is still upheld in that particular instant. Similarly, this has to be performed for all instants. In our case we use an SMT solver to do this, in particular we are using Yices[86] SMT solver. We export the clock relations, any user defined clock constraints, initial type assignments and the inferred types as an SMT formula. This serves as the preamble for the objective defined later on. An objective in our case is the goal of the user expressed as SMT formula. Some of the examples of goals are as follows,

1. For the given type assignments, under what constraints the model is type consistent?
2. Under what constraints, the model is always type consistent?
3. Are there any constraints which will break the type consistency of the model?

For each of these goals, the SMT formula differs. We append the objective to the preamble and ask the SMT solver to check if the objective is satisfiable under the constraints expressed in the preamble. The SMT solver answers with *sat* or *unsat* result indicating satisfiable or unsatisfiable. For both these solutions, the SMT solver also gives a counter example, which can be easily translated into invariants or constraints. Next we illustrate how we can achieve goals 1 and 2 for the model shown in Figure 38.

(i) The clock relations obtained from the hierarchical clock relation graph for the model are as below.

$$clk_u = clk_w = clk_v = clk_y$$

$$clk_z = clk_x \vee clk_y$$

$$clk_w = clk_z \wedge clk_c \wedge c$$

(ii) The initial type assignments provided by the user are given below.

$$d_x = \langle LT^{-1}, msec^{-1}, clk_x \rangle = T1$$

$$d_u = \langle LT^{-2}, msec^{-2}, clk_u \rangle = T2$$

(iii) The type assignments obtained from the inference algorithm are as below.

$d_w = d_v = d_y = T2$

$d_z = T1@clk_x/T2@((not\ clk_x) \wedge clk_y)$

$d_w = T1/T2@([\hat{c}])$

Initially we declare the variables and the function prototypes to be used later on in the SMT formula as shown in the declaration section of Listing 3.7.1. We then export the constraints from (i), (ii) and (iii) as the SMT formula as shown in Listing 3.7.1's preamble section.

Listing 3.7.1 SMT formula to check Type Consistency

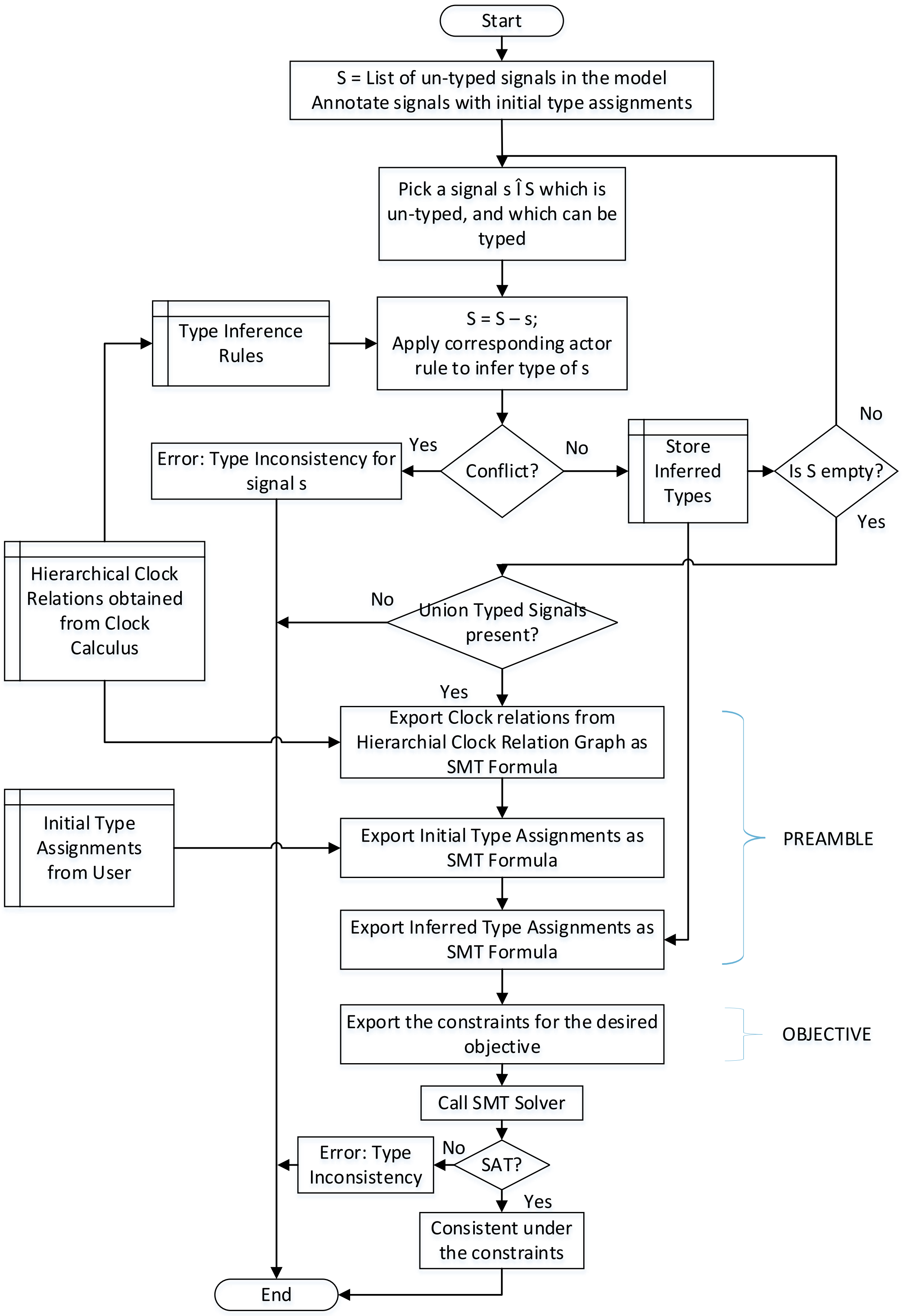
```
  ;; DECLARATIONS : Data types
(define-type DATATYPE (scalar T1 T2))

;; DECLARATIONS : Data type variable names
(define-type VARNAME (scalar d_x d_y d_z d_w d_v d_u))

;; DECLARATIONS : Function that maps VARNAME to DATATYPE
(define typeof::(-> VARNAME DATATYPE))

;; DECLARATIONS : Clock variables
(define clk_x::bool)
(define clk_y::bool)
(define clk_z::bool)
(define clk_c::bool)
(define clk_w::bool)
(define clk_u::bool)
(define clk_v::bool)
(define c::bool)

;; PREAMBLE : Relations from hierarchial clock relation graph
(assert (ite (or clk_x clk_y) clk_z (not clk_z)))
(assert (= clk_y clk_v))
(assert (ite (and clk_z clk_c (= c true)) clk_w (not clk_w)))
(assert (= clk_w clk_u))
(assert (= clk_w clk_v))
(assert (or clk_x clk_y clk_z clk_w clk_u clk_v))

;; PREAMBLE : Initial type assignments
(assert (= (typeof d_x) T1))
(assert (= (typeof d_u) T2))

;; PREAMBLE : Inferred types
(assert (= (typeof d_v) T2))
(assert (= (typeof d_w) T2))
(assert (= (typeof d_y) T2))
(assert (ite (and clk_z clk_c (= c true)) (= (typeof d_w) T2)
            (= 0 0)))
(assert (ite clk_x (= (typeof d_z) T1) (= 0 0) ))
(assert (ite (and (not clk_x) clk_y) (= (typeof d_z) T2) (= 0 0) ))

;; OBJECTIVE : Constraints from type inference rules
(assert (ite clk_v (= (typeof d_v) (typeof d_u)) (= 0 0) ))
(assert (ite clk_w (= (typeof d_w) (typeof d_u)) (= 0 0) ))
(assert (ite clk_v (= (typeof d_v) (typeof d_w)) (= 0 0) ))
(assert (ite clk_y (= (typeof d_y) (typeof d_v)) (= 0 0) ))
(assert (ite (and clk_z clk_c c) (= (typeof d_w) (typeof d_z))
            (= 0 0)))
(assert (ite clk_x (= (typeof d_z) (typeof d_x)) (= 0 0) ))
(assert (ite (and (not clk_x) clk_y) (= (typeof d_z) (typeof d_y))
            (= 0 0)))

(check)
;; Show constraints that satisfies the objective
(show-model)
```

Now let us consider the goal 1. The objective is to obtain the constraints such that the model is type consistent with the current preamble. For the model to be type consistent, it has to satisfy all the type inference rules. We export the rules for each of the actor as the objective function. This is shown in the objective section of Listing 3.7.1. This formula is then given to Yices SMT solver and Listing 3.7.2 shows the output of the solver.

Listing 3.7.2 Output of Yices Solver

```
sat
(= clk_z true)
(= clk_u true)
(= c true)
(= clk_w true)
(= clk_v true)
(= clk_c true)
(= clk_x false)
(= clk_y true)
(function typeof
  (type (-> VARNAME DATATYPE))
  (= (typeof d_x) T1)
  (default T2))
```

The output of SMT solver indicates that the objective can be satisfied when signals clk_z, clk_u, clk_w, clk_v, clk_c, clk_y are all $true$, $clk_x = false$, $c = true$ and the dimensions of all other signals except x is $T2 = [LT^{-2}]$, while signal x has the dimension LT^{-1}. Interpreting this result, we can say that, the model is type consistent with the current type assignments, signals z, u, c, w, v, y are all present, signal x is absent and the signal c carries $true$ value. Under these conditions, the types of all other signals except signal x is $T2$, while the type of x is $T1$. Note that, this is not the only possible constraint under which the model is type consistent. This is one of the possible constraints.



Start
S = List of un-typed signals in the model
Annotate signals with initial type assignments
Pick a signal s Î S which is un-typed, and which can be typed
Type Inference Rules
S = S – s;
Apply corresponding actor rule to infer type of s
Error: Type Inconsistency for signal s
Yes
Conflict?
No
Store Inferred Types
Is S empty?
No
Yes
Hierarchical Clock Relations obtained from Clock Calculus
No
Union Typed Signals present?
Yes
Export Clock relations from Hierarchial Clock Relation Graph as SMT Formula
Initial Type Assignments from User
Export Initial Type Assignments as SMT Formula
Export Inferred Type Assignments as SMT Formula
PREAMBLE
Export the constraints for the desired objective
OBJECTIVE
Call SMT Solver
Error: Type Inconsistency
No
SAT?
Yes
Consistent under the constraints
End

Figure 40: Flowchart for Type Inference and SMT Analysis

Now let us consider the goal 2. To ensure that the model is always type consistent, we have to obtain all possible constraints under which type consistency is preserved. To do so, we can add the obtained constraint back as a preamble and ask the SMT solver to provide newer constraints. Once the SMT solver returns an *unsat* solution, it indicates that there are no more newer constraints. Then we can say that as long as any of those constraints are satisfied individually, the model remains type consistent (goal 2).

Figure 40 shows the complete flowchart for the type inference and SMT analysis.

3.7.3 Summary

Dimensional and unit inconsistencies in critical embedded software could result in a failure of the entire system. Often it is hard to ensure consistency by verification as the software is not annotated with the units and dimensions information. On the contrary, software design approaches based on models provide easy annotation options and could be formally verified. Our framework allows the user to specify the type information for some signals in the model and further infer the type information for all signals. Most model-based designs allow signals of union types, which makes, checking for correctness difficult. Our work has addressed this problem by using the concept of clocks of the corresponding signals. Our proposed approach is generic enough to be employed in most of the model-based design tools. Since our analysis is statically done on the models, the generated software need not do any dynamic checks and hence its run-time efficiency will not suffer. Also, our proposed approach is sound but not complete. The analysis for union types depends on the clock relations of the signals. Clock relations that are derived from other clocks based on certain conditions, could be complicated expressions and might not always be possible to be evaluated. As a result of this, we resort to abstraction and over-approximation which might lead to the incorrect rejection of accurate specifications.

3.8 SMT based Value Range Analysis of Polychronous models

With the increasing amount of software used in safety critical systems, it is absolutely essential to ensure that programs produce expected output values for every possible run. These expected values are the values that satisfies pre-specified constraints on the outputs and they guarantee that system is behaving as expected. For smaller systems, this can be achieved by doing exhaustive simulation and verifying the outputs. As systems become complex, exhaustive simulation is very difficult and may not be possible. Traditionally, verifying such complex system's software is done using static analysis techniques. Complex systems are first abstracted and a simpler model is derived with just enough information needed to do the analysis. Some of the prominent disadvantages of this approach are,

(i) difficulties involved in automating the abstraction procedure and
(ii) establishing the behavioral equivalence between abstracted model and the original system. Alternatively, it is much easier to build the model of a complex

system ground-up, perform analysis on the model and then use the same model for generating code for the complex system. In Section 3.8, we propose range inference rules for the polychronous modeling language - MRICDF, which are further used to derive range constraints on the signals. Further, we also propose a technique that converts these range constraints to SMT constraints and verifies signal value range related properties on the model.

Figure 41 shows an MRICDF model for a simple automated bathtub system. For readability purposes, it's corresponding SIGNAL code is also provided. Despite being small, this example is sufficient to illustrate the problem statement.

This model implements a bathtub system where the level of water is automatically controlled. This model has no inputs and 1 output -- *level*. Output *'level'* indicates the level of water in the bathtub at any given instant of time. *'level'* is computed based on two other signals – *'faucet'* and *'pump'*, which are used to increase and decrease the amount of water in the tub respectively. The objective of this system is to ensure that bathtub is never empty and it never overflows. In other words, value of *level* is always in safe range: *0 < level < 9*.

To confirm that the system satisfies the objective, we need to know –

(i) Will level <=0 ever?

(ii) Will level >=9 ever?

SIGNAL program for Simple Automated Bathtub Embedded System

```
process BathTub =
(?
  ! integer level;)
(| level      := zlevel + faucet - pump
  | zlevel    := level init $1
  | faucet    := zfaucet + (1 when zlevel<=4)
  | zfaucet := faucet init $0
  | pump        := zpump + (1 when zlevel>=7)
  | zpump      := pump init $0
   |)
  where
      integer zLevel, zfaucet, zpump, faucet, pump;
  end
```

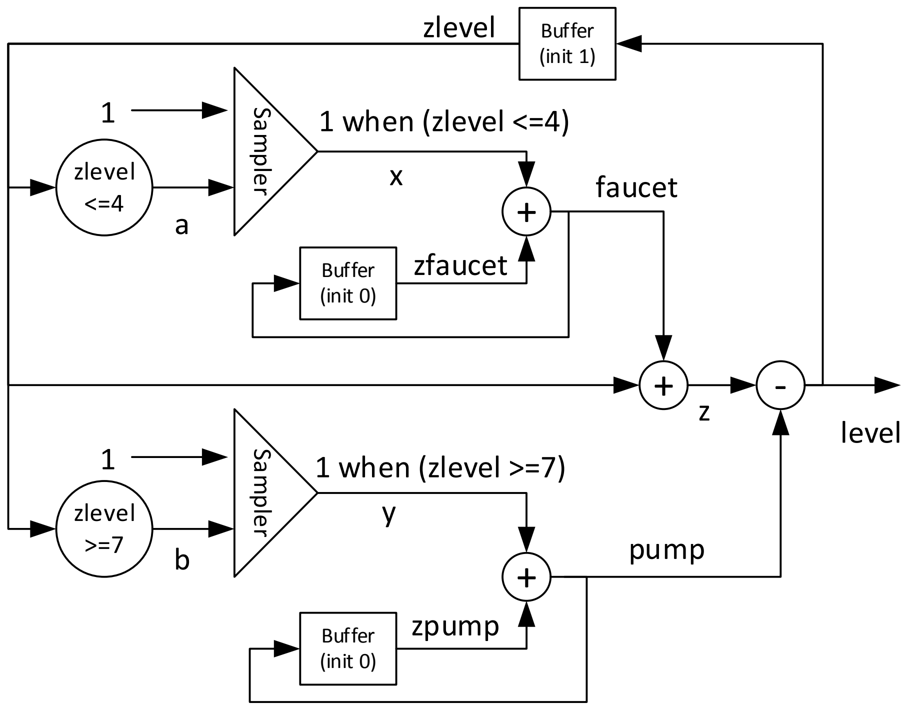


Figure 41: Example MRICDF model and SIGNAL program for Simple Automated Bathtub Embedded System

To answer these questions, one has to perform value range analysis on the model. Let *P* be the program that implements the model in Figure 41. Let the set P(level) = {l_1, l_2, ..., l_n} represent all the values *'level'* can take during all possible executions of *P*. We simply represent the set of all such possible values using a closed interval as below.

$$P(level) = [l_{level}, h_{level}]$$

where, l_{level} and h_{level} are the lowest and the highest values that $level$ can take. In other words,

$$l_{level} \le value(level) \le h_{level},$$

where $value(level) \in \{l_1, l_2, ..., l_n\}$.

Note that, $level$ need not take all the values present in the interval. It might only take few, but it can't take any value outside the interval. Value range analysis techniques help us determine the range $[l_{level}, h_{level}]$, which can further be used to verify properties of the system.

Numerous research works such as [66], [70], [36], etc., have explored solutions to the problem of signal value range analysis in modeling languages that have synchronous model of time. However, we cannot adopt these solutions to modeling languages with polychronous model of time, at-least not directly as polychronous languages pose additional complications. The complications arise due to the fact that,

(i) the signals are polychronous in nature – this means, a signal may or may not be present during any particular reaction;

(ii) Priority merge construct merges two signals with priority to first one. If the ranges for the two input signals are not same, it will result in split intervals for outputs, which will require a "widening" operation to be performed on the intervals;

(iii) Interval widening operations in case of polychronous signals should consider clocks of the signals too. This implies that clock calculus must be an integral part of this approach;

(iv) Also, interval widening operations can easily result in trivial intervals (such as $[\infty,-\infty]$). Thus, it is also important to narrow the intervals during analysis by proposing sound interval narrowing operations.

Our major contributions described in Section 3.8 are,

• We propose range inference rules for the actors of polychronous language -MRICDF. Given value range constraints for the input signals, the range inference rules help us to statically infer the value range constraints for the rest of the signals in the model.

• During inferring range constraints for priority merge actor, the range of the output of the merge actor might be a union of multiple ranges. We propose a variant of the interval "widening" operation that approximates and merges the split ranges into a single range while considering the clocks of the input signals. We also propose a variant of interval "narrowing" operation that uses the clock constraints from the model and tries to restrict the widened ranges.

• Further, we propose a technique to convert all the value range constraints into SMT constraints and allow the user to verify signal value range related properties.

In the past, there have been other similar approaches proposed for C, C++, Java and other synchronous languages. We believe that the polychronous model of time adds additional complications and makes our extensions and the inference rules novel. Also, the SMT-based verification technique – especially considering the clock calculus constraints makes it unique.

The rest of Section 3.8 is structured as follows. Section 3.8.1 describes inference rules for MRICDF actors and the algorithm which uses the rules and infers range constraints of the signals. We also describe the interval "widening" and "narrowing" operations. In Section 3.8.2, we explain with an example on how we can use the inferred range constraints and an SMT solver to verify properties related to value ranges.

3.8.1 Range Inference Rules and Analysis

Modeling languages such as MRICDF, SIGNAL, etc are typically used to model control systems. As explained in Section 3.2, an MRICDF model is an interconnected network of both primitive and composite actors. Each of the connections represent a physical signal in the control system. Each of these physical signals carry values corresponding to some physical quantity and has additional attributes such as dimensions, units etc. In this work, we focus on the "value" attribute of the signal. Our proposed extension to EmCodeSyn, allows the users to specify the values a signal can take in terms of *intervals* (a.k.a *range*). Further, the inference algorithm can infer *ranges* for the rest of the signals using the range inference rules. In this research effort, we restrict ourselves to Integer and Boolean value range analysis. However, the same analysis can be extended to floating point too.

3.8.1.1 Definitions

Let Z denote the set of all integers. We denote the range of a signal x as R(x) and is defined as

$$\begin{aligned} R(x) &= [l_x, h_x], \\ &\quad where\ l_x \le h_x\ and\ l_x, h_x \in Z \end{aligned}$$

If the range of a signal is unknown, we denote it using $\perp$ and is defined as,

$$\perp = [?,?]$$

If we are unable to infer the range constraints of a signal, we say that its range is the maximum range T. It is defined as,

$$\mathrm{T} = [-\infty, \infty]$$

In theory, $\perp$ refers to an unknown range, while T refers to the full range. Initially, all the signals except the input signals will have their ranges set to unknown range. If we cannot determine the range constraints of any signal, we say its range is full range and proceed.

Based on these definitions, we now define the lattice of ranges as

$$\begin{aligned} L &= \{\perp\} \cup \{[l,h]\} \cup \{\mathrm{T}\}, \\ &\quad where\ l \le h\ and\ l, h \in Z \end{aligned}$$

This lattice is partially ordered by the$\sqsubseteq$ such that,

$$\perp \sqsubseteq r,\ \ \forall r \in L$$

$$[l_1, h_1] \sqsubseteq [l_2, h_2],\quad if l_2 \le l_1 \le h_1 \le h_2$$

$$[l, h] \sqsubseteq T,\quad \forall\ \ l, h \in Z$$

Structurally, the lattice can be represented as shown in Figure 42. The arrows in Figure 42 represent the$\sqsubseteq$relation that was defined earlier.

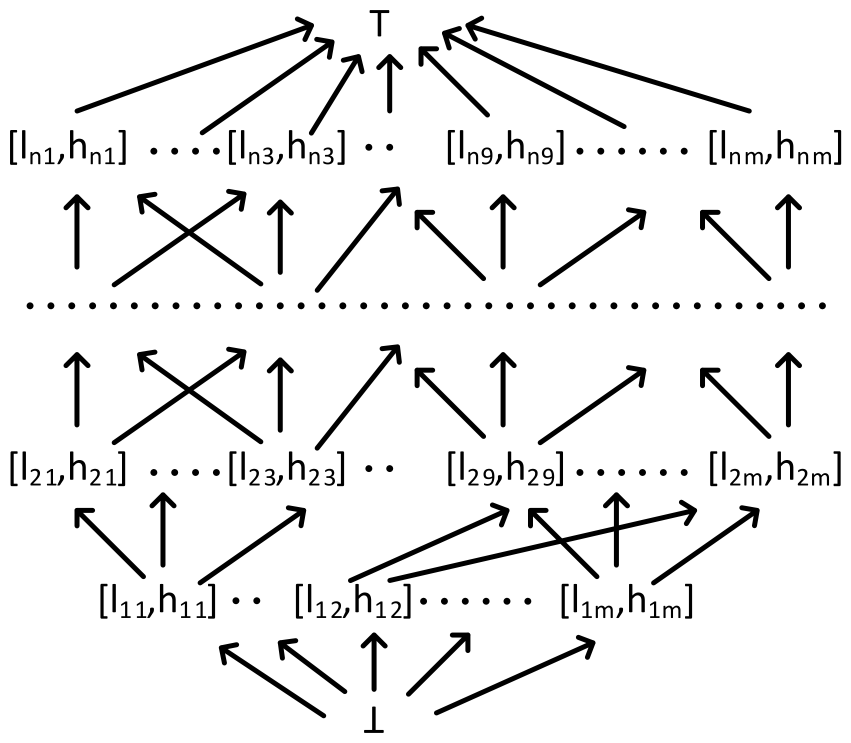
3.8.1.2 Widening operation (∐)

Widening operator (∐) takes two input ranges and soundly approximates into a single output range using the principle of convex approximation.

3.8.1.2.1 Case 1: Boolean Sets

Boolean signals can have only *true* or *false* as their value. Hence the widening operation on Boolean signals is defined on the set of values rather than intervals,

$$\begin{aligned} \{true\} \amalg \{true\} &= \{true\} \\ \{false\} \amalg \{false\} &= \{false\} \\ \{true, false\} \amalg \{true\} &= \{true, false\} \\ \{true, false\} \amalg \{false\} &= \{true, false\} \\ \{true\} \amalg \{false\} &= \{true, false\} \end{aligned}$$



⊤
[ln1,hn1] . . . . [ln3,hn3] . . [ln9,hn9] . . . . . . [lnm,hnm]
. . . .
[l21,h21] . . . . [l23,h23] . . [l29,h29] . . . . . . [l2m,h2m]
[l11,h11] . . [l12,h12] . . . . . . [l1m,h1m]
⊥

Figure 42: Lattice Structure

3.8.1.2.2 Case 2: Integer Intervals

For integer intervals, the widening operation is defined as,

$$
\begin{aligned}
[l_1, h_1]ò[l_2, h_2] &= [l_3, h_3] \\
&\quad \textit{where}\, [l_3, h_3] = W(l_1, l_2, h_1, h_2)
\end{aligned}
$$

Function W depends on the actual operation performed on the input signals. For arithmetic operations the definition of W would be as below,

$$
\begin{aligned}
Op+ : W(l_1, l_2, h_1, h_2) &= [l_1 + l_2, h_1 + h_2] = [l_3, h_3] \\
Op- : W(l_1, l_2, h_1, h_2) &= [l_1 - h_2, l_2 - h_1] = [l_3, h_3] \\
Op\times : W(l_1, l_2, h_1, h_2) &= [l_3, h_3],\, where \\
l_3 &= min(l_1 l_2, l_1 h_2, h_1 l_2, h_1 h_2) \\
h_3 &= max(l_1 l_2, l_1 h_2, h_1 l_2, h_1 h_2) \\
Op\div : W(l_1, l_2, h_1, h_2) &= [l_3, h_3],\, where \\
l_3 &= min(\frac{l_1}{l_2}, \frac{l_1}{h_2}, \frac{h_1}{l_2}, \frac{h_1}{h_2}) \\
h_3 &= max(\frac{l_1}{l_2}, \frac{l_1}{h_2}, \frac{h_1}{l_2}, \frac{h_1}{h_2}) \\
&\quad and\ 0 \in [l_2, h_2]
\end{aligned}
$$

Other intelligent widening operations require more complex lattice structures such as anti-interval lattice, congruence lattice, polyhedras, etc, or a combination of them. In this work, we restrict ourselves to interval lattices. Unrestricted widening in interval lattices could easily result in signal ranges becoming trivial ($\top$). Thus, whenever we are doing a widening operation, we check if we can restrict the widening by using the constraints on the clocks. This is explained in the next section.

3.8.1.3 Narrowing Operation

During clock calculus step, a lot of sub-clocks are defined. Often these sub-clocks are derived by constraining other known clocks and those constraints could be based on the values of signals. If such constraints are predicated on the values of the input signals of the actor, then we could use such constraints while inferring range constraints of its output. An example of how such narrowing operation could be done while inferring range constraints for output of Sampler actor is shown below. Consider a sampler actor whose definition is,

$$o = Sampler(i, j);$$

Let us assume that,

$$\begin{aligned} R(i) &= [-\infty, \infty] \\ R(j) &= \{true, false\} \\ j &= true, when\ (-200 \le i \le 200) \end{aligned}$$

Let us also assume that $\hat{i}$ is same as $\hat{j}$, in other words, whenever signal i is present signal j is also present, but its value can be $true$ or $false$. If we employ Equation 11 for inferring the range of output, we get,

$$R(o) = [l_o, h_o] = [-\infty, \infty]$$

But, theoretically the output can never take a value beyond the range $[-200, 200]$, because whenever i is beyond $[-200, 200]$, the sampling signal j has the value $false$. We can improve the accuracy of range analysis by intersecting the previously inferred range $R(o)$ with the range constraints obtained from clock calculus $[-200, 200]$. This will result in,

$$R(o)_{new} = R(o) \cap [-200, 200] = [-200, 200]$$

Similarly, we could potentially use other such constraints obtained from clock calculus to further restrict the range of output signals. Of course, this type of opportunistic narrowing operations cannot be done always, but when possible, they can be done very easily.

3.8.1.4 Value Range Inference Rules

MRICDF language has two polychronous actors – *merge* and *sampler*. The output of these actors not only depends on the values of its inputs, but it also depends on their clocks. Thus, it is essential that we consider the clocks of signals during inferring value ranges. For this, we introduce the concept of "tagged range" of a signal. It is defined as the range of a signal x at some clock $\hat{c}$ and is denoted as $R(x)@\hat{c}$. If the clock c is $false$, in other words, the signal c is absent, it means $R(x)@\hat{c}$ is $\perp @ \phi$. Based on the this and the definitions in the previous section, we now introduce the range inference rules for the primitive actors of MRICDF.

Let i, j and o be any three signals whose clocks are $\hat{i}$, $\hat{j}$ and $\hat{o}$ respectively. Let $R(i)$, $R(j)$ and $R(o)$ be the ranges of signals i, j and o respectively. Then, we can define the tagged ranges as below,

$$\begin{aligned} R(i)@\hat{i} &= [l_i, h_i]@\hat{i} \\ R(j)@\hat{j} &= [l_j, h_j]@\hat{j} \\ R(o)@\hat{o} &= [l_o, h_o]@\hat{o} \end{aligned}$$

3.8.1.4.1 i. Buffer Actor

Operation: o = Buffer(i) init iv

If the initial value of the Buffer actor is subsumed in the range of the input, then, the range of the output of Buffer actor is same as its input range. Else the output range has to be widened to include the initial value.

$$\frac{value(i) \in R(i)@\hat{\imath}, \quad InitialValue \in [iv, iv]}{R(o) = [l_o, h_o] = R(i) \coprod [iv, iv] = [l_i, h_i] \coprod [iv, iv]}$$

The inference rule states that, if i is present, if $R(i)$ is its range and the initial value is 'iv', the range of output o at clock $\hat{o}$ denoted by $R(o)@\hat{o}$ is defined as below,

$$R(o)@\hat{o} = ([l_i, h_i] \coprod [iv, iv])@\hat{\imath}$$

3.8.1.4.2 ii. Sampler Actor

Operation: o = Sampler(i,j)

The output of Sampler actor is same as its first input (i) when both the inputs are present and the second input (j also known as sampling input) is $true$. To derive the inference rules for sampler actor, we define a clock $[\hat{j}]$, which is always $true$, when the sampling input j is present and its value is $true$. Based on this, we derive the inference rules for sampler actor as below,

$$\frac{value(i) \in R(i)@\hat{i}, \; value(j) \in \{true, \; false\}@\hat{j}}{R(o) = [l_o, h_o] = R(i) \; when \; (\hat{i} \wedge [\hat{j}]) \; is \; true}$$

The inference rule states that if input i is present and its range is denoted by $R(i)$, and the sampling input j is present and $true$, then we can say that the range of output o is same as $R(i)$. Thus, we can say that -

$$R(o)@\hat{o} = [l_i, h_i]@(\hat{i} \wedge [\hat{j}])$$

Note that, if we know from clock calculus that $[\hat{j}] \subseteq \hat{i}$, in other words, i is always present whenever j is present and it has a value $true$, we can simplify the above equation as

$$R(o)@\hat{o} = [l_i, h_i]@\hat{i},\ if\ [\hat{j}] \subseteq \hat{i}$$

Also, if we know that $[\hat{j}] \wedge \hat{i} = \phi$, in other words, i is never present when j is present and it has value $true$ or j never has a value $true$ when i is present, then we can simply equation 10 as below,

$$R(o)@\hat{o} = \perp @\phi,\ if\ [\hat{j}] \wedge \hat{i} = \phi$$

3.8.1.4.3 iii. Merge Actor

Operation: o = Merge(i,j)

The output of the merge actor is first input (when it is present) or second input (when the first input is absent and second input is present). The merge actor does not produce any output when both its inputs are absent. Inference rule for merge actor is as below,

$$\frac{value(i) \in R(i)@\hat{\imath},\quad value(j) \in R(j)@\hat{\jmath}}{R(o) = [l_o, h_o] = R(i)@\hat{\imath} \amalg R(j)@(\hat{\jmath} \wedge !\,\hat{\imath})}$$

If $R(i)$ô$R(j)$, we can simplify the inference rule for merge actor as,

$$\frac{value(i) \in R(i)@\hat{i},\ value(j) \in R(j)@\hat{j}}{R(o) = [l_o, h_o] = R(i)@(\hat{i} \vee \hat{j} \wedge !\hat{i})}$$

From the clock calculus, if we also know that $\hat{j} \subseteq \hat{i}$, then we can simplify the inference rule for merge actor further to,

$$R(o) = [l_o, h_o] = R(i)@\hat{i}$$

One can further extend the simplification procedure based on additional clock constraints.

3.8.1.4.4 iv. Function Actor

Operation: o = Function(i,j) = i $\bigoplus$ j

The output of function actor depends on not just its inputs but also on the operation ($\bigoplus$) implemented by the function actor. Thus, to derive the inference rules for function actor, a formal description of the operation ($\bigoplus$) implemented is essential. Using the formal description, we can then infer the range of the output. For simpler operations such as $\bigoplus \in \{+,-.\times,\div,||,\&\&\}$, it is easier to derive the ranges using the widening operation rules (Eq 7) as below,

$$\frac{value(i) \in R(i)@\hat{\imath}, \quad value(j) \in R(j)@\hat{\jmath}}{R(o) = [l_o, h_o] = (R(i) \coprod R(j))@\hat{\imath}}$$

But for more complex operations, the user would need to provide a formal description of the operation or the user can directly enter the range of outputs.

For composite actors, we recursively dive inside each of them and use the above rules to infer the ranges and propagate them to the upper level.

In the next section, we explain how to use these inferred range constraints to verify properties related to signal value ranges.

3.8.2 SMT based Verification of properties related to Signal Value Ranges

In Section 3.8.2, we propose a technique to convert the range constraints obtained by applying the inference rules to SMT constraints. Further, we can use the SMT constraints for various analysis including property verification. We use the example model in Figure 41 as a running example to explain this section. We assume that the reader is familiar with Yices [86] SMT language constructs.

3.8.2.1 SMT constraints for Constants

The output of a constant actor is always a single value. For example, in our running example we have 2 constant actors ($c1$, $c2$) always outputting 1 as the value. Thus, we can say that -

$$R(c1) = [l_{c1}\ h_{c1}] = [1,1]$$

This can be rewritten in SMT language as shown below,

(assert (= l_c1 h_c1))

3.8.2.2 SMT constraints for Sampler actors

From Equation 10, we know that the range of output of function actor is same as its input when both clocks are present and the sampling input is $true$. For illustration purposes, let us consider the sampler actor that computes the value x in our example model.

$$x = Sampler(c1, a)$$

Using Equation 10 and knowing that $c1 = 1, \hat{c}1 = true$ always, we can say that

$$R(x) = [1,1]\ when\ (true \wedge [\hat{a}]) = true$$

This can be expressed in SMT language as

(assert (ite (= clk_a true) (and (= l_x l_c1) (= h_x h_c1)) (= 0 0)))

3.8.3 SMT constraints for Merge actors

Using the inference rule for merge actor ($o = Merge(i, j)$), we can say that,

$$R(o) = R(i)@\hat{\imath} \coprod R(j)@(\hat{\jmath} \wedge !\,\hat{\imath})$$

We can convert this to SMT constraint as below,

(assert (ite (= clk_i true) (and (= l_o l_i) (= h_o h_i)) (= 0 0)))
(assert (ite (= (and (not clk_i) clk_j) true) (and (= l_o l_j) (= h_o h_j)) (= 0 0)))

In our example model, we do not have any Merge actors.

3.8.4 SMT constraints for Function actors

We know from Equation 14, that the output ranges of function actor depends on the operation performed. In our example model, we have have a total of 6 function actors out of which 4 are performing '$+$' operation and the other 2 are comparators. To illustrate the steps to derive SMT constraints for function actors, we choose one of the function actor that performs the following operation -

$$z = Function(l_{faucet}, l_{zlevel}) = l_{faucet} + l_{zlevel}$$

Using Equation 14 and 7 we can say that the output range of the above function actor is the sub of the input ranges.

$$R(z) = R(zlevel) + R(faucet)$$

This can be expressed in Yices SMT language as shown below,

(assert (and (= (+ l_faucet l_zlevel) l_z) (= (+ h_faucet h_zlevel) h_z)))
(assert (= (+ faucet zlevel) z))

3.8.5 SMT constraints for Buffer actors

As explained in the earlier section, Buffer actor may introduce recursive constraints while deriving range inference constraints. This will be handled implicitly by the inference algorithm. Thus, while deriving the SMT constraints for Buffer actor, we do not express the dependency of the range of output on its input. We now consider the Buffer actor that produces $zlevel$. We know that,

$$\begin{aligned} zlevel &= Buffer(level)\ init\ 1,\ and \\ R(zlevel) &= [l_{zlevel}\ h_{zlevel}] \\ &= ([l_{level}, h_{level}] \grave{o} [iv, iv]) \\ &\quad where\ iv = 1 \end{aligned}$$

This means the value of $zlevel$ is either initial value(1) or $level$. Also the value of $zlevel$ is either initial value(1) or falls within range $[l_{zlevel}\ h_{zlevel}]$. We can express these constraints using the SMT formula shown below.

(assert (or (= zlevel 1) (= zlevel level)))
(assert (or (= zlevel 1) (and (<= l_zlevel zlevel) (<= zlevel h_zlevel))))

3.8.6 Case Study: Automated Bathtub System

We now show how to employ the steps described above and convert the range constraints of the MRICDF model in Figure 41. Initially, we declare all the clock and range variables that will be used later on. In the next part of the SMT formula, we append the constraints obtained from clock calculus and the trivial clock relation that ensures that there is at-least one signal active in each valid reaction. We then append the constraint for basic definition of ranges - the value of a signal always reside between its minimum and maximum bounds. Finally, we convert the range constraints between input and output ranges of each actor to SMT constraints and append them. Listing 3.8.1 shows the constructed SMT formula for the Automated Bathtub example shown in the MRICDF model in Figure 41.

Listing 3.8.1: SMT formula for MRICDF model in Figure 41

```
;; DECLARATIONS : Clock variables
(define clk_level::bool)
(define clk_zlevel::bool)
(define clk_faucet::bool)
(define clk_zfaucet::bool)
(define clk_pump::bool)
(define clk_zpump::bool)
(define clk_a::bool)
(define clk_b::bool)
(define clk_x::bool)
(define clk_y::bool) (define clk_z::bool)

;; DECLARATIONS : Range variables
(define level::int)
(define l_level::int)
(define h_level::int)
(define zlevel::int)
(define l_zlevel::int)
(define h_zlevel::int)
(define faucet::int)
(define l_faucet::int)
(define h_faucet::int)
(define zfaucet::int)
(define l_zfaucet::int)
(define h_zfaucet::int)
(define pump::int)
(define l_pump::int)
(define h_pump::int)
(define zpump::int)
(define l_zpump::int)
(define h_zpump::int)
(define z::int)
(define l_z::int)
```

```
(define h_z::int)
(define a::bool)
(define b::bool)
(define x::int)
(define l_x::int)
(define h_x::int)
(define y::int)
(define l_y::int)
(define h_y::int)

;; Constraints from clock calculus
(assert (= clk_level clk_zlevel))    (assert (= clk_faucet clk_zfaucet))
(assert (= clk_a clk_zlevel))            (assert (= clk_b clk_zlevel))
(assert (= clk_x clk_zfaucet))           (assert (= clk_x clk_faucet))
(assert (= clk_faucet clk_zlevel)) (assert (= clk_faucet clk_z))
(assert (= clk_y clk_zpump))              (assert (= clk_y clk_pump))
(assert (= clk_pump clk_level))        (assert (= clk_pump clk_z))
(assert (= clk_pump clk_zpump))        (assert (= clk_z clk_level))
(assert (ite (>= zlevel 7) (= b true) (= 0 0)))
(assert (ite (= b true) (= clk_b true) (= 0 0)))
(assert (ite (= clk_b true) (= clk_y true) (= 0 0)))
(assert (ite (<= zlevel 4) (= a true) (= 0 0)))
(assert (ite (= a true) (= clk_a true) (= 0 0)))
(assert (ite (= clk_a true) (= clk_x true) (= 0 0)))

;;Trivial Clock Relation
(assert (or clk_level clk_zlevel clk_faucet clk_zfaucet clk_pump clk_zpump clk_a
clk_b clk_x clk_y clk_z))

;; Basic relations from definition of ranges
(assert (and (<= l_level level) (<= level h_level)))
(assert (and (<= l_faucet faucet) (<= faucet h_faucet)))
(assert (and (<= l_pump pump) (<= pump h_pump)))
(assert (and (<= l_x x) (<= x h_x)))
(assert (and (<= l_y y) (<= y h_y)))
(assert (and (<= l_z z) (<= z h_z)))

;; SMT Constraints for constants
(assert (= l_c1 h_c1)) (assert (= l_c2 h_c2))

;; SMT Sampler actor constrains
(assert (ite (= clk_a true) (and (= l_x 1) (= h_x 1) (= 0 0)))
(assert (ite (= clk_b true) (and (= l_y 1) (= h_y 1)) (= 0 0)))
;; SMT Function (+) actor constraints
(assert (and (= (+ l_faucet l_zlevel) l_z) (= (+ h_faucet h_zlevel) h_z)))
(assert (= (+ faucet zlevel) z))
(assert (and (= (+ l_y l_zpump) l_pump) (+ h_y h_zpump) h_pump)))
(assert (= (+ y zpump) pump))
(assert (and (= (+ l_x l_zfaucet) l_faucet) (= (+ h_x h_zfaucet) h_faucet)))
(assert (= (+ x zfaucet) faucet))
(assert (and (= (+ l_z l_pump) l_level) (= (+ h_z h_pump) h_level)))
(assert (= (+ pump z) level))

;; SMT Buffer actor
(assert (or (= zlevel 1) (= zlevel level)))
(assert (or (= zfaucet 0) (= zfaucet faucet)))
(assert (or (= zpump 0) (= zpump pump)))
(assert (or (= zlevel 1) (and (<= l_zlevel zlevel) (<= zlevel h_zlevel))))
(assert (or (= zfaucet 0) (and (<= l_zfaucet zfaucet) (<= zfaucet h_zfaucet))))
(assert (or (= zpump 0) (and (<= l_zpump zpump) (<= zpump h_zpump))))
(push)
```

At the start of Section 3.8, we wanted to know if,

- Can $level <= 0$ ever?
- Can $level >= 9$ ever?

We now show how to answer these questions. First, we rewrite these properties as SMT constraints as shown in Listing 3.8.2. We then append the new SMT constraints to the previously derived SMT formula for the model shown in Listing 3.8.1 and run Yices [86] SMT solver on the appended SMT formula. We get the result as *unsat* for both the properties. This indicates that these properties can never be satified for the MRICDF model in Figure 41.

Listing 3.8.2: Properties to verify on the MRICDF model in Figure 41

```
;; Property to verify: Can level<=0
(assert (<= level 0))
;; Check for satisfiability
(check)

;; ANS
unsat

;; Property to verify: Can level>9
(pop)
(assert (> level 9))
;; Check for satisfiability
(check)

;; ANS
unsat
```

As a sanity check, we wanted to see if the property – "$level > 0$" is satisfied by MRICDF model in Figure 41. We checked it by appending the SMT constraints shown in Listing 3.8.3 to the SMT formula shown in Listing 3.8.1 and calling Yices SMT solver on the appended formula. The solver returned sat for this SMT formula indicating that the property $level > 0$ is indeed satisfied by the MRICDF model of Figure 41.

Listing 3.8.3: Verifying property "level > 0" for the MRICDF model in Figure 41

```
;; Property to verify: Can level>0
(assert (> level 0))
;; Check for satisfiability
(check)

;; ANS
sat
```

3.8.3 Summary

Software that produces unexpected outputs in safety critical applications could be catastrophic (Ariane 5 crash [21]). Ensuring that the software only produces expected outputs by means of exhaustive simulation is not scalable and sometimes it may not even be possible. In case of model-based design approaches, one way to ensure this is by doing value range analysis on the models. In Section 3.8, we proposed sound techniques to perform value range analysis of MRICDF models. Our techniques are generic enough, that they can be employed in other polychronous languages too. We also showed how to export the constraints obtained during range analysis as SMT constraints and prove properties related to signal value ranges.

In this research effort, to keep the analysis less complicated, we restricted ourselves to Integer and Boolean intervals for the signal values. In future, we would like to explore floating point intervals. We only considered widening operations with respect to simple interval lattice. In future, we plan to explore other complex lattices such as congruence, anti-interval, polyhedras, etc. This will also lead to more intelligent widening operations.

4 Conclusions and Future Work

Over the past few decades, the size and complexity of the safety critical embedded systems have increased tremendously. This has presented unique challenges for the design approaches that used to design and develop such complex systems. The nature of these systems being complex and safety critical, would require the design approaches to not just scale , but also provide guarantees regarding the correctness. Traditional design approaches are not easily scalable and they generally require extensive testing to provide correctness guarantees. Numerous alternatives to traditional design approaches have been proposed that can tackle these challenges. One such approach is Formal Model Driven Design (MBD) and Development. MBD design approaches are being increasingly adopted in the industry due to various advantages they offer over the traditional design approaches. In our research work, we have explored MBD based software synthesis techniques, hardware synthesis techniques and verification and validation techniques. These techniques are scalable, sound and generic enough that they can be adopted in other MBD approaches.

4.1 Conclusions

4.1.1 Software Synthesis techniques

Automated, error free, deterministic software synthesis is one of the key advantages that formal model-based design approaches offer over the traditional approaches. There are numerous MBD tools being developed by academic researchers and industries that provides the ability to synthesize sequential code from synchronous specifications, but not many tools to synthesize code from polychronous specifications. Previous attempts at code synthesis from polychronous specifications (MRICDF models) ([43], [40]) were also specifically targeted at sequential code synthesis. We concentrated our research efforts towards synthesis of multi-threaded code from MRICDF models. We proposed a novel Boolean theory based approach for determining if a given MRICDF model is concurrently implementable or not. Our Boolean theory involves generation of prime implicates using SMT solvers. We proposed a notion of partial triggers and proposed a technique to infer partial triggers from the prime implicates. Further, we proposed technique to identify the synchronization constraints between the partial triggers. We then proposed a code generation technique by mapping the partial triggers to threads. We performed scalability and performance analysis of the proposed technique. For the considered benchmarks, we noticed that the performance of the synthesized multi-threaded code was about 18% slower than the performance of the hand-written multi-threaded code. Performance analysis revealed a few bottle necks that was causing the dip in the performance of the synthesized multi-threaded code. One of them was - excessive synchronizations. We proposed a novel technique based on analysis of affine clocks that identifies all the avoidable synchronizations and removes them from the synthesized code, which in-turn improves the performance of the multi-threaded code.

4.1.2 Hardware Synthesis techniques

In [55], the authors explained how Conditional Partial Order Graphs (CPOGs) enable us to compactly and efficiently describe and store instruction sets. Further, they explained how they can be used to identify parallelisms and synthesize custom instruction sets. On the same line of thought, we proposed a technique that accepts formal MRICDF/SIGNAL [43] specifications and compiles them to Conditional Partial Order Graphs (CPOGs). These CPOGs are further used to generate custom instruction sets for Application Specific Instruction set Processors (ASIPs).

4.1.3 Verification and Validation techniques

One of the pre-requisites for an MRICDF model to be sequentially or concurrently implementable is that, it should not contain any causal loops. In the past, numerous solutions have been proposed for doing causality analysis. However, most of these approaches only work on Boolean abstraction of the predicates. This may lead to sound, but imprecise decisions being made, which in-turn may lead to erroneously rejecting an MRICDF model to be non-synthesizable. We proposed an SMT and Polyhedra based approach for performing causality analysis which considers both Boolean and Integer predicates. Our proposed approach helps in making better decisions while performing causal analysis. Further, we also proposed an approach to identify the constraints under which the causality behavior of the system is exhibited. Then, we explained how these constraints can be used to generate a wrapper which would always keep the system in safe operating region.

Case studies in [61] showed us that units and dimensional inconsistencies between signals at the interfaces could result in catastrophic failures. In Section 3.7, we explained why performing dimensional analysis on the code is much harder and why it makes sense to perform the analysis on the models itself. We proposed an novel SMT based approach for performing unit and dimensional analysis statically on the polychronous models. To the best of our knowledge, this is the first ever approach for performing dimensional analysis on polychronous languages. The main advantage of our approach is that it considers the clock constraints of the signals which checking for dimensional consistencies. Our approach is scalable and adds minimum overhead.

Software used in safety critical embedded system is required to produce expected output values for every possible run. By conducting static value range analysis on the program, one can check if the signals ever take any values out of some pre-defined bound. There are approaches proposed in the past for doing value range analysis for synchronous programming languages such as C/C++/Java. But, the polychronous model of computation brings in additional complications which would require the value range analysis techniques to consider the clocks of the signals along with their values. We proposed a novel SMT based technique to perform value range analysis in polychronous languages and explained it with a case study. Our proposed approach considers the clocks of the signals too.

4.2 Future Work

4.2.1 Software Synthesis techniques

In our initial proposed approach for multi-threaded code synthesis, the clock tree construction and the code generator implementation are done targeting accuracy and not efficiency of the generated code. To improve efficiency of the generated code, one can apply optimization transformations on the clock tree which can help in generating a better load-balanced code. Also, our proposed technique for mapping of partial triggers to threads might not be efficient, especially if the amount of work done by the thread is not substantially large than thread creation and destruction overhead. Thus, as another optimization step, one can create a thread pool and map partial triggers to tasks – adopting the concept of Intel Threading Building Blocks (TBB). One can also improve the code generator to synthesize a more cache friendly multi-threaded code. Our second approach for multi-threaded code synthesis based on analysis for affine relations between clocks could be easily extended to synthesize code for GPUs (Graphical Processing Units) or similar hardware accelerators.

4.2.2 Hardware Synthesis techniques

In this report, we explored how ASIPs could be synthesized from MRICDF models by compiling them to CPOGs. An interesting future work could be to explore techniques for co-synthesis of hardware and software. Exploring the aspect of sequential and concurrent implementability by applying transformations on the CPOGs could be useful.

4.2.3 Verification and Validation techniques

The proposed causality analysis technique does not account for dynamic behavior of variables. The current polyhedral library we use, Polylib, is restricted to integer and approximate floating point constraints expressed as linear system of inequalities and equations. This is a restriction on the library and not on the technique we proposed. In future we plan to use a different library which can handle floating point constraints. We also want to expand the analysis beyond polyhedra into non-linear system of inequalities and equations.

Our proposed dimensional analysis technique is sound but not complete. The analysis for union types depends on the clock relations of the signals. Clock relations that are derived from other clocks based on certain conditions, could be complicated expressions and might not always be possible to be evaluated. As a result of this, we resort to abstraction and over-approximation which might lead to the incorrect rejection of accurate specifications. An interesting future work would be improve this.

In our value range analysis technique, to keep the analysis less complicated, we restricted ourselves to Integer and Boolean intervals for the signal values. A future work, could be to explore floating point intervals. Also, we only considered widening operations with respect to simple interval lattice. As future work, one can explore complex lattices such as congruence, anti-interval, polyhedras, etc. This will also lead to more intelligent widening operations.

A more ambitious future work is extending the type system of the Polychronous language - MRICDF. This can help in generating invariants for a system, detecting overflow in signals, refining the causal analysis by considering ranges, etc.

4.2.4 EmCodeSyn Tool development

The usability of a software tool plays as much a crucial role as the functionality of the tool in its promotion and adoption among new users. We have made deep strides to improve the usability of the EmCodeSyn tool by separating front-end with back-end. We plan to constantly add new functionalities and improve the EmCodeSyn tool. Going forward, we would like to provide better support for navigating, composing MRICDF models. The earlier versions of EmCodeSyn were cross platform compatible, but the latest one isn't. In future, we would like to explore the options of making it cross platform compatible.

5 References

[1] X. Amatriain. An object-oriented metamodel for digital signal processing with a focus on audio and music: A brief catalogue of graphical moc’s, http://xavier.amatriain.net/thesis/html/node37.html, 2004.

[2] M. Anand, I. Lee, G. Pappas, and O. Sokolsky. Unit and dynamic typing in hybrid systems modeling with charon. In *2006 IEEE Conf on Computer Aided Control System Design*, pages 56 –61, oct. 2006.

[3] C. Andre. Representation and analysis of reactive behaviors: A synchronous approach. 1996.

[4] C. Andre. Syncharts: A visual representation of reactive behavior, technical report rr-95-52, 13s, 1995.

[5] E. S. G. at the Technical University of Kaiserslautern. Averest framework, http://www.averest.org/, 2014.

[6] R. Bagnara, P. M. Hill, and E. Zaffanella. Applications of polyhedral computations to the analysis and verification of hardware and software systems. *Theoretical Computer Science*, 410(46), 2009.

[7] D. Baudisch, J. Brandt, and K. Schneider. Multithreaded code from synchronous programs: Extracting independent threads for OpenMP. In *Design, Automation and Test in Europe*, Dresden, Germany, 2010.

[8] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation 1,2,3. *Inf. Comput.*, 163:125–171, November 2000.

[9] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64 – 83, January 2003.

[10] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Sci. Comput. Program.*, 16:103–149, Sept. 1991.

[11] G. Berry, M. Kishinevsky, and S. Singh. System level design and verification using a synchronous language. In *Computer Aided Design, 2003. ICCAD-2003. International Conference on*, pages 433–439, Nov 2003.

[12] G. Berry. The esterel v5 language primer - version 5.21, 1999.

[13] F. Besson, T. Jensen, and J.-P. Talpin. Polyhedral analysis for synchronous languages. In *Static Analysis: Proc. of the 6th Int. Sym, vol 1694 of Lecture Notes in Computer Science*, pages 51–68. Springer-Verlag, 1999.

[14] F. Boussinot. Reactive c: An extension of c to program reactive systems, 1991.

[15] B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. 15th European Symposium on Programming, ESOP 2006, pages 264–278, 2006.

[16] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, Sept. 1994.

[17] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[18] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL ’78, pages 84–96, New York, NY, USA, 1978. ACM.

[19] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of the 5th Symposium on Principles of programming languages*, New York, NY, USA, 78. ACM.

[20] J. J. Downs and E. F. Vogel. A plant-wide industrial process control problem. *Computers & Chemical Engineering*, 17(3):245–255, Mar. 1993.

[21] M. Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84–, Mar. 1997.

[22] J. Dunfield and F. Pfenning. Type assignment for intersections and unions in call-by-value languages. FOSSACS’03/ETAPS’03, pages 250–266, Berlin, Heidelberg, 2003. Springer-Verlag.

[23] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009.

[24] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.

[25] S. A. E. et. al. Cec: The columbia esterel compiler - version 0.4, www.cs.columbia.edu/ sedwards/cec/, 2012.

[26] P. Feiler and D. de Niz. Assip study of real-time safety-critical embedded software-intensive system engineering practices,, 2008.

[27] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI ’02, pages 234–245, New York, NY, USA, 2002. ACM.

[28] C. for Hybrid and B. Embedded Software Systems (CHESS), University of California. Ptolemy project - heterogeneous modeling and design, http://ptolemy.eecs.berkeley.edu/ptolemyii/, 2002.

[29] A. Gamatie and L. Gonnord. Static analysis of synchronous programs in signal for efficient design of multi-clocked embedded systems. In *Proc. of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES ’11, pages 71–80, New York, NY, USA, 2011. ACM.

[30] S. J. Garland and N. Lynch. Foundations of component-based systems. pages 285–312, 2000.

[31] O. Grumberg, A. Schuster, and A. Yadgar. Memory efficient all-solutions sat solver and its application for reachability analysis. In *Formal Methods in Computer-Aided Design*, volume 3312, pages 275–289. Springer Berlin Heidelberg, 2004.

[32] P. L. Guernic, T. Gautier, M. L. Borgne, and C. L. Maire. Programming real-time applications with SIGNAL. *Proc. of The IEEE*, 79:1321–1336, 1991.

[33] P. L. Guernic, P. L. Guernic, J.-P. Talpin, J. pierre Talpin, J.-C. L. Lann, J. christophe Le Lann, and P. Espresso. Polychrony for system design. *Journal for Circuits, Systems and Computers*, 12:261–304, 2002.

[34] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305 –1320, September 1991.

[35] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[36] T. M. Inc. Polyspace bug finder, http://www.mathworks.com/ products/polyspace-bug-finder/, 2014.

[37] INRIA. The esterel language - inria, http://www-sop.inria.fr/meije/esterel/esterel-eng.html, 2014.

[38] M. Jain, M. Balakrishnan, and A. Kumar. Asip design methodologies: survey and issues. In *VLSI Design, 2001. Fourteenth International Conference on*, pages 76–81, 2001.

[39] B. A. Jose, A. Gamatie, M. Kracht, and S. K. Shukla. Improved false causal loop detection in polychronous specification of embedded software, fermat technical report 2011-08.

[40] B. A. Jose, A. Gamatie, J. Ouy, and S. Shukla. Smt based false causal loop detection during code synthesis from polychronous specifications. In *MEMOCODE Conference Proceedings*, July 2011.

[41] B. A. Jose, H. D. Patel, S. K. Shukla, and J.-P. Talpin. Generating multi-threaded code from polychronous specifications. *Electron. Notes Theor. Comput. Sci.*, 238, June 2009.

[42] B. Jose, J. Pribble, L. Stewart, and S. Shukla. Emcodesyn: A visual framework for multi-rate data flow specifications and code synthesis for embedded applications. In *Forum on Specification Design Languages*, pages 1–6, sept. 2009.

[43] B. Jose and S. Shukla. An alternative polychronous model and synthesis methodology for model-driven embedded software. In *15th ASPDAC,*, Jan. 2010.

[44] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[45] A. J. Kennedy and A. J. Kennedy. Relational parametricity and units of measure. In *In 24th ACM Symp. on Principles of Programming Languages*, pages 442–455. ACM Press, 1997.

[46] A. Kountouris and C. Wolinski. A method for the generation of hdl code at the rtl level from a high-level formal specification language. In *Circuits and Systems,. Proceedings of the 40th Midwest Symposium on*, volume 2, pages 1095–1098, Aug 1997.

[47] A. Kountouris and C. Wolinski. Hierarchical conditional dependency graphs as a unifying design representation in the codesis high-level synthesis system. In *The 13th International Symposium on System Synthesis*, pages 66–71, 2000.

[48] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, Jan. 1987.

[49] M.-K. Leung, T. M. 0002, E. A. Lee, E. Latronico, C. P. Shelton, S. Tripakis, and B. Lickly. Scalable semantic annotation using lattice-based ontologies. In A. SchÃ¼rr and B.

Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 393–407. Springer, 2009.

[50] B. Lickly, C. Shelton, E. Latronico, and E. A. Lee. A practical ontology framework for static model analysis. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, pages 23–32, New York, NY, USA, 2011. ACM.

[51] O. Maffes and P. L. Guernic. Distributed implementation of signal: Scheduling & graph clustering. In *Proc. of the* 3^{rd} *Int. Sym. on Formal Techniques in Real-Time and Fault-Tolerant Systems*, London, UK, 94.

[52] F. Maraninchi and Y. Remond. Argos: an automaton-based synchronous language, 2001.

[53] F. Maraninchi. The argos language: Graphical representation of automata and description of reactive systems. In *In IEEE Workshop on Visual Languages*, 1991.

[54] Mathworks. Hdl coder : Generate verilog and vhdl code for fpga and asic designs, http://www.mathworks.com/products/hdl-coder.

[55] A. Mokhov, D. Sokolov, M. Rykunov, and A. Yakovlev. Formal modelling and transformations of processor instruction sets. In *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, pages 51–60, 2011.

[56] A. Mokhov and A. A. Yakovlev. Conditional partial order graphs: Model, synthesis, and application. *IEEE Transactions on Computers*, 59(11):1480–1493, 2010.

[57] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[58] P. K. Murthy and E. A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50:3306–3309, 2002.

[59] M. Nanjundappa, M. Kracht, J. Ouy, and S. K. Shukla. Synthesizing embedded software with safety wrappers through polyhedral analysis in a polychronous framework. In *ESLsyn'12*, pages 24 –29, june 2012.

[60] M. Nanjundappa and S. Shukla. Verification of unit and dimensional consistencies in polychronous specifications. In *Specification Design Languages, 2014. FDL 2014. Forum on*, pages 1 –6, October. 2014.

[61] T. R. of the University of California Davis. Case studies: Metric/english conversion errors, 2011.

[62] V. Papailiopoulou, D. Potop-Butucaru, Y. Sorel, de Simone R., L. Besnard, and J. Talpin. From design-time concurrency to effective implementation parallelism: The multi-clock reactive case. In *ESLsyn'11*, june 2011.

[63] C. A. Petri. Phd thesis - kommunikation mit automaten, 1962.

[64] D. Potop-Butucaru, Y. Sorel, R. de Simone, and J.-P. Talpin. From concurrent multi-clock programs to deterministic asynchronous implementations. *Fundam. Inf.*, 108(1-2):91–118, Jan. 2011.

[65] F. Remondino and N. Borlin. Polylib - a library of polyhedral functions. In *Int. Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, Vol. XXXIV, H.-G. Maas and D. Schneider (Eds)*, 2004.

[66] R. E. Rodrigues. Scalable and precise range analysis on the interval lattice, http://homepages.dcc.ufmg.br/$\widetilde{\mathrm{f}}$ernando/ publications/books/raphael_msc.pdf, 2014.

[67] P. Roy and N. Shankar. SimCheck: An expressive type system for Simulink. In C. Muñoz, editor, *Proc of the 2nd NASA Formal Methods Symposium*, pages 149–160, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.

[68] K. Schneider, J. Brandt, and E. Vecchie. Efficient code generation from synchronous programs. *ACM International Conf on Formal Methods and Models for Co-Design*, pages 165–174, 2006.

[69] K. Schneider. The synchronous programming language quartz v2.0, http://es.cs.uni-kl.de/publications/datarsg/schn09.pdf, 2010.

[70] A. Simon. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[71] S. Singh. Hardware/software synthesis and verification using esterel. In A. A. McEwan, S. A. Schneider, W. Ifill, and P. H. Welch, editors, *CPA*, volume 65 of *Concurrent Systems Engineering Series*, pages 371–378. IOS Press, 2007.

[72] I. Smarandache, T. Gautier, and P. Le Guernic. Validation of mixed signal-alpha real-time systems through affine calculus on clock synchronisation constraints. In *FMâ€™99 â€” Formal Methods*, volume 1709, pages 1364–1383, 1999.

[73] I. Smarandache and P. Le Guernic. Affine transformations in signal and their application in the specification and validation of real-time systems. In

Transformation-Based Reactive Systems Development, volume 1231, pages 233–247, 1997.

[74] M. Strecker. Clock type soundness in synchronous languages, technical report, 2009.

[75] S. Suhaib, B. Jose, S. Shukla, and D. Mathaikutty. Formal transformation of a kpn specification to a gals implementation. In *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pages 84–89, Sept 2008.

[76] J.-P. Talpin, J. Ouy, L. Besnard, and P. L. Guernic. Compositional design of isochronous systems. *Design, Automation and Test Conference*, 0:928–933, 2008.

[77] E. Technologies. Scade suite by esterel technologies, http://www.esterel-technologies.com/, 2014.

[78] S. Yuan, L. H. Yoong, and P. S. Roop. Compiling esterel for multi-core execution. In *DSD*, pages 727–735, 2011.

[79] Labview - laboratory virtual instrument engineering workbench - national instruments, www.ni.com/support/labview/.

[80] Modelica and the modelica association - modelica association, https://www.modelica.org/.

[81] Polychrony: A toolset for signal, http://www.irisa.fr/espresso/polychrony.

[82] Scilab - xcos, https://www.scilab.org/scilab/gallery/xcos.

[83] Simulink - simulation and model-based design - the mathworks inc, http://www.mathworks.com/products/simulink/.

[84] Stateflow - model and simulate decision logic using state machines and flow charts - the mathworks inc, http://www.mathworks.com/products/stateflow/.

[85] Unified modeling language (uml), http://www.uml.org/.

[86] The yices smt solver - b. dutertre and l. de moura, http://yices.csl.sri.com/.

Appendix

Example: Generated Multi-threaded Code

Consider the MRICDF model as shown in Figure 43.

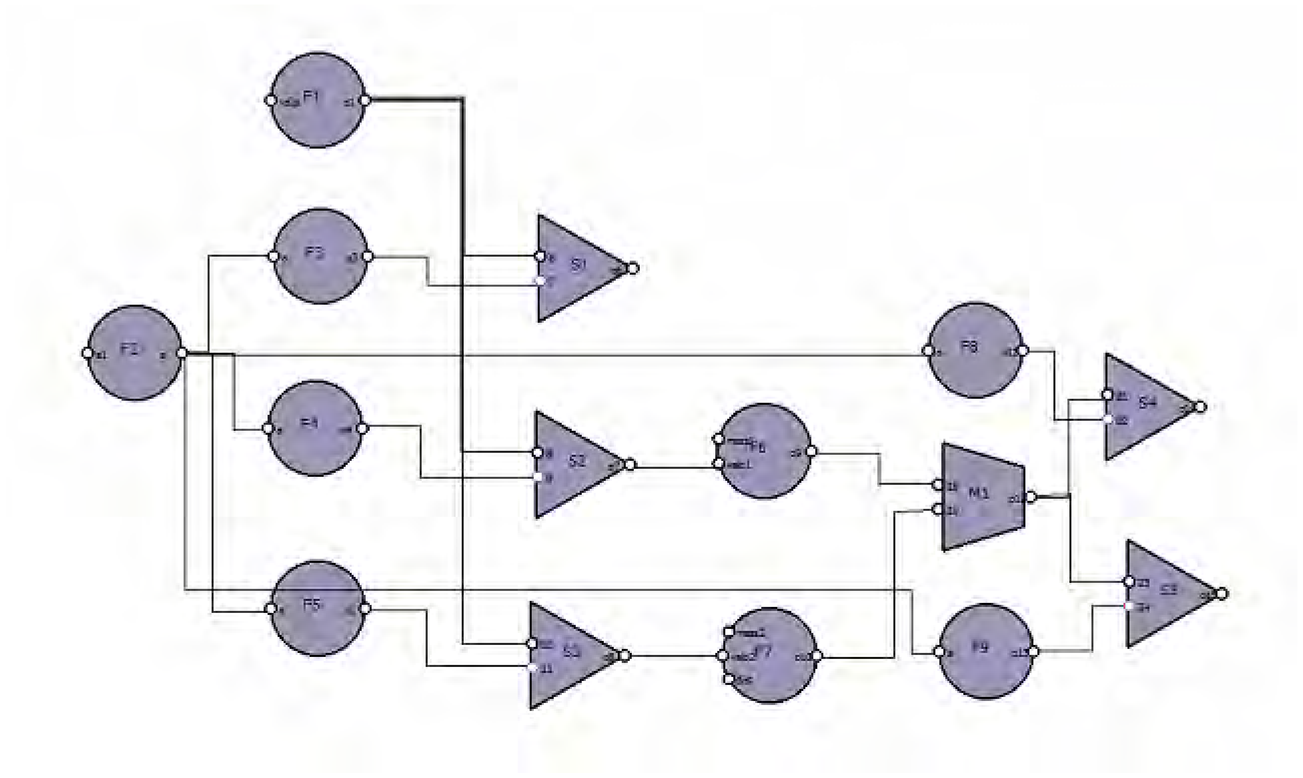


Figure 43: Energy Meter Model

The multi-threaded code generated for the model shown in Figure 43 is shown below.

```
/****************************************************************************/
/* * energyMeter_mt_main.cpp */
/****************************************************************************/
#include <iostream>
#include <fstream>
#include <pthread.h>
#include "MRICDFlib.cpp"
#include "energyMeter_lib.h"

#define MAXINSTANCES -1

ofstream F_S1o6;
ofstream F_S4o14;
ofstream F_S5o15;

SigQueue<int> SQ_S2i8;
SigQueue<bool> SQ_S2i9;
```

```
SigQueue<int> SQ_S3i10;
SigQueue<bool> SQ_S3i11;
SigQueue<int> SQ_M1i18;
SigQueue<int> SQ_M1i19;
SigQueue<int> SQ_S5i23;
SigQueue<bool> SQ_S5i24;
SigQueue<int> SQ_S4i21;
SigQueue<bool> SQ_S4i22;
SigQueue<int> SQ_S1i6;
SigQueue<bool> SQ_S1i7;

void* block1(void *arg){

        int instance = 0;
        ifstream F_vaIp;
        Sig<int> vaIp;
        Sig<int> F1o1;

        F_vaIp.open("vaIp.txt", ifstream::in);
        if(!F_vaIp.is_open()){
                cout <<"Error: File vaIp.txt does not exist"<< endl;
                pthread_exit(NULL);
        }

        while(instance != MAXINSTANCES){

                readLine(vaIp,F_vaIp);
                vaIp.setInstance(instance);

                if( F_vaIp.eof() ){
                        F_vaIp.close();
                        SQ_S1i6.write_finish();
                        SQ_S2i8.write_finish();
                        SQ_S3i10.write_finish();
                        pthread_exit(NULL);
                }

                F1(vaIp,F1o1);

                if(F1o1.clk()){
                        SQ_S1i6.write(F1o1);
                }
                if(F1o1.clk()){
                        SQ_S2i8.write(F1o1);
                }
                if(F1o1.clk()){
                        SQ_S3i10.write(F1o1);
                }
                instance++;
        }
}

void* block2(void *arg){
```

```
int instance = 0;
ifstream F_a1;
Sig<int> a1;
Sig<int> a;
Sig<bool> F9o13;
Sig<bool> F8o11;
Sig<bool> F4o4;
Sig<bool> F5o5;
Sig<bool> F3o3;

F_a1.open("a1.txt", ifstream::in);
if(!F_a1.is_open()){
        cout <<"Error: File a1.txt does not exist"<< endl;
        pthread_exit(NULL);
}

while(instance != MAXINSTANCES){

        readLine(a1,F_a1);
        a1.setInstance(instance);

        if( F_a1.eof() ){
                F_a1.close();
                SQ_S5i24.write_finish();
                SQ_S4i22.write_finish();
                SQ_S2i9.write_finish();
                SQ_S3i11.write_finish();
                SQ_S1i7.write_finish();
                pthread_exit(NULL);
        }

        F2(a1,a);
        F9(a,F9o13);
        F8(a,F8o11);
        F4(a,F4o4);
        F5(a,F5o5);
        F3(a,F3o3);

        if(F9o13.clk()){
                SQ_S5i24.write(F9o13);
        }
        if(F8o11.clk()){
                SQ_S4i22.write(F8o11);
        }
        if(F4o4.clk()){
                SQ_S2i9.write(F4o4);
        }
        if(F5o5.clk()){
                SQ_S3i11.write(F5o5);
        }
        if(F3o3.clk()){
                SQ_S1i7.write(F3o3);
                }
                instance++;
        }
}

void* block3(void *arg){

        int instance = 0;
        Sig<int> S2i8;
        Sig<bool> S2i9;
        Sig<int> S2o7;
        ifstream F_mass1;
        Sig<int> mass1;
        Sig<int> F6o9;

        F_mass1.open("mass1.txt", ifstream::in);
        if(!F_mass1.is_open()){
                cout <<"Error: File mass1.txt does not exist"<< endl;
                pthread_exit(NULL);
        }

        while(instance != MAXINSTANCES){
                S2i8 = SQ_S2i8.read();
                S2i9 = SQ_S2i9.read();

                if(S2i8.is_finished() || S2i9.is_finished()){
                        SQ_M1i18.write_finish();
                        pthread_exit(NULL);
                }

                Sampler(S2i8,S2i9,S2o7);

                readLine(mass1,F_mass1);
                mass1.setInstance(instance);

                if( F_mass1.eof() ){
                        F_mass1.close();
                        SQ_M1i18.write_finish();
                        pthread_exit(NULL);
                }

                F6(mass1,S2o7,F6o9);

                SQ_M1i18.write(F6o9);
                instance++;
        }
}

void* block4(void *arg){

        int instance = 0;
        Sig<int> S3i10;
        Sig<bool> S3i11;
```

```
Sig<int> S3o8;
ifstream F_mass2;
Sig<int> mass2;
ifstream F_dist;
Sig<int> dist;
Sig<int> F7o10;

F_mass2.open("mass2.txt", ifstream::in);
if(!F_mass2.is_open()){
        cout <<"Error: File mass2.txt does not exist"<< endl;
        pthread_exit(NULL);
}
F_dist.open("dist.txt", ifstream::in);
if(!F_dist.is_open()){
        cout <<"Error: File dist.txt does not exist"<< endl;
        pthread_exit(NULL);
}

while(instance != MAXINSTANCES){
        S3i10 = SQ_S3i10.read();
        S3i11 = SQ_S3i11.read();

        if(S3i10.is_finished() || S3i11.is_finished()){
                SQ_M1i19.write_finish();
                pthread_exit(NULL);
        }

        Sampler(S3i10,S3i11,S3o8);

        readLine(mass2,F_mass2);
        mass2.setInstance(instance);

        if( F_mass2.eof() ){
                F_mass2.close();
                F_dist.close();
                SQ_M1i19.write_finish();
                pthread_exit(NULL);
        }

        readLine(dist,F_dist);
        dist.setInstance(instance);

        if( F_dist.eof() ){
                F_mass2.close();
                F_dist.close();
                SQ_M1i19.write_finish();
                pthread_exit(NULL);
        }

        F7(mass2,S3o8,dist,F7o10);

        SQ_M1i19.write(F7o10);
        instance++;
        }
}

void* block5(void *arg){

        int instance = 0;
        Sig<int> M1i18;
        Sig<int> M1i19;
        Sig<int> M1o12;

        while(instance != MAXINSTANCES){
                M1i18 = SQ_M1i18.read();
                M1i19 = SQ_M1i19.read();

                if(M1i18.is_finished() || M1i19.is_finished()){
                        SQ_S4i21.write_finish();
                        SQ_S5i23.write_finish();
                        pthread_exit(NULL);
                }

                Merge(M1i18,M1i19,M1o12);

                if(M1o12.clk()){
                        SQ_S4i21.write(M1o12);
                }
                if(M1o12.clk()){
                        SQ_S5i23.write(M1o12);
                }
                instance++;
        }
}

void* block6(void *arg){

        int instance = 0;
        Sig<int> S5i23;
        Sig<bool> S5i24;
        Sig<int> S5o15;

        while(instance != MAXINSTANCES){
                S5i23 = SQ_S5i23.read();
                S5i24 = SQ_S5i24.read();

                if(S5i23.is_finished() || S5i24.is_finished()){
                        pthread_exit(NULL);
                }

                Sampler(S5i23,S5i24,S5o15);

                if(F_S5o15.is_open()){
                        if(S5o15.clk()){
                                F_S5o15 << S5o15 << endl;
                        }
                } else {
                        cerr <<"Error: Output File S5o15.txt is not open."<< endl;
                        pthread_exit(NULL);
                }
                instance++;
        }
}

void* block7(void *arg){

        int instance = 0;
        Sig<int> S4i21;
        Sig<bool> S4i22;
        Sig<int> S4o14;


        while(instance != MAXINSTANCES){
                S4i21 = SQ_S4i21.read();
                S4i22 = SQ_S4i22.read();

                if(S4i21.is_finished() || S4i22.is_finished()){
                        pthread_exit(NULL);
                }

                Sampler(S4i21,S4i22,S4o14);

                if(F_S4o14.is_open()){
                        if(S4o14.clk()){
                                F_S4o14 << S4o14 << endl;
                        }
                } else {
                        cerr <<"Error: Output File S4o14.txt is not open."<< endl;
                        pthread_exit(NULL);
                }
                instance++;
        }
}

void* block8(void *arg){

        int instance = 0;
        Sig<int> S1i6;
        Sig<bool> S1i7;
        Sig<int> S1o6;


        while(instance != MAXINSTANCES){
                S1i6 = SQ_S1i6.read();
                S1i7 = SQ_S1i7.read();

                if(S1i6.is_finished() || S1i7.is_finished()){
                        pthread_exit(NULL);
                }
                Sampler(S1i6,S1i7,S1o6);

                if(F_S1o6.is_open()){
                        if(S1o6.clk()){
                                F_S1o6 << S1o6 << endl;
                        }
                } else {
                        cerr <<"Error: Output File S1o6.txt is not open."<< endl;
                        pthread_exit(NULL);
                }
                instance++;
        }
}


intmain(int argc, char *argv[]){

        //Open Output Files
        F_S1o6.open("S1o6.txt");
        F_S4o14.open("S4o14.txt");
        F_S5o15.open("S5o15.txt");

        pthread_t *threads;
        pthread_attr_t attr;

        threads = (pthread_t *)malloc(8*sizeof(pthread_t));
        pthread_attr_init(&attr);

        pthread_create(&threads[0], &attr, block1, (void *)0);
        pthread_create(&threads[1], &attr, block2, (void *)0);
        pthread_create(&threads[2], &attr, block3, (void *)0);
        pthread_create(&threads[3], &attr, block4, (void *)0);
        pthread_create(&threads[4], &attr, block5, (void *)0);
        pthread_create(&threads[5], &attr, block6, (void *)0);
        pthread_create(&threads[6], &attr, block7, (void *)0);
        pthread_create(&threads[7], &attr, block8, (void *)0);

        for(int i=0;i<8;i++){
                pthread_join(threads[i], NULL);
        }

        //Close Output Files
        F_S1o6.close();
        F_S4o14.close();
        F_S5o15.close();

        return0;
}
/*****************************************************************************
***********/
```

```
/****************************************************************************
***********/
/* energyMeter_lib.cpp
/****************************************************************************
***********/

#include "energyMeter_lib.h"


voidF1( const Sig<int>& F1i1, Sig<int>& F1o1){

        F1o1.setInstance(F1i1.getInstance());
        if(F1i1.clk()){

                F1o1.setClk();

                F1o1.funcVal(vaInput_F1(F1i1.val()));

        } else {
                F1o1.clearClk();
        }
}

intvaInput_F1(int vaIp){
return vaIp;
}

voidF3( const Sig<int>& F3i3, Sig<bool>& F3o3){

        F3o3.setInstance(F3i3.getInstance());
        if(F3i3.clk()){

                F3o3.setClk();

                F3o3.funcVal(Sig3_F3(F3i3.val()));

        } else {
                F3o3.clearClk();
        }
}

boolSig3_F3(int a){
        int x = 7*a+28;
        int y = 27*a-17;

        if((x >= 42) || (y <37))
                returntrue;
        else
                returnfalse;
}

voidF4( const Sig<int>& F4i4, Sig<bool>& F4o4){
        F4o4.setInstance(F4i4.getInstance());
        if(F4i4.clk()){

                F4o4.setClk();

                F4o4.funcVal(Sig2_F4(F4i4.val()));

        } else {
                F4o4.clearClk();
        }
}

boolSig2_F4(int a){
        if(a >= 2)
                returntrue;
        else
                returnfalse;
}

voidF5( const Sig<int>& F5i5, Sig<bool>& F5o5){

        F5o5.setInstance(F5i5.getInstance());
        if(F5i5.clk()){

                F5o5.setClk();

                F5o5.funcVal(Sig1_F5(F5i5.val()));

        } else {
                F5o5.clearClk();
        }
}

boolSig1_F5(int a){
        if(-a > -2)
                returntrue;
        else
                returnfalse;
}

voidF6( const Sig<int>& F6i12, const Sig<int>& F6i13, Sig<int>& F6o9){

        F6o9.setInstance(F6i12.getInstance());
        if(F6i12.clk() && F6i13.clk()){

                F6o9.setClk();

                F6o9.funcVal(computeMomemtum_F6(F6i12.val(),F6i13.val()));

        } else {
                F6o9.clearClk();
        }
}
```

```
intcomputeMomemtum_F6(int mass1, int velo1){
        return mass1*velo1;
}

voidF7( const Sig<int>& F7i14, const Sig<int>& F7i15, const Sig<int>& F7i16,
Sig<int>& F7o10){

        F7o10.setInstance(F7i14.getInstance());
        if(F7i14.clk() && F7i15.clk() && F7i16.clk()){

                F7o10.setClk();


        F7o10.funcVal(computeEnergy_F7(F7i14.val(),F7i15.val(),F7i16.val()));

        } else {
                F7o10.clearClk();
        }
}

intcomputeEnergy_F7(int mass2, int velo2, int dist){
        return mass2*velo2*dist;
}

voidF9( const Sig<int>& F9i20, Sig<bool>& F9o13){

        F9o13.setInstance(F9i20.getInstance());
        if(F9i20.clk()){

                F9o13.setClk();

                F9o13.funcVal(Sig5_F9(F9i20.val()));

        } else {
                F9o13.clearClk();
        }
}

boolSig5_F9(int a){
int x = 27*a - 17;

if(x<37)
returntrue;
else
returnfalse;
}

voidF8( const Sig<int>& F8i17, Sig<bool>& F8o11){

        F8o11.setInstance(F8i17.getInstance());
        if(F8i17.clk()){
                F8o11.setClk();

                F8o11.funcVal(Sig4_F8(F8i17.val()));

        } else {
                F8o11.clearClk();
        }
}

boolSig4_F8(int a){
int x = 7*a + 28;

if(x>=42)
returntrue;
else
returnfalse;
}

voidF2( const Sig<int>& F2i2, Sig<int>& F2o2){
        F2o2.setInstance(F2i2.getInstance());

        if(F2i2.clk()){
                F2o2.setClk();
                F2o2.funcVal(inputA_F2(F2i2.val()));
        } else {
                F2o2.clearClk();
        }
}

intinputA_F2(int a1){
return a1;
}

/*****************************************************************************
/

/*****************************************************************************
/
/* energyMeter_lib.cpp */
/*****************************************************************************
/


#ifndef ENERGYMETER_LIB_H
#define ENERGYMETER_LIB_H

#include "MRICDFlib.cpp"
#include <omp.h>

#define NUMTHREADS omp_get_num_procs()

usingnamespace std;
```

```
voidF1( const Sig<int>& F1i1, Sig<int>& F1o1);
intvaInput_F1(int vaIp);


voidF3( const Sig<int>& F3i3, Sig<bool>& F3o3);
boolSig3_F3(int a);


voidF4( const Sig<int>& F4i4, Sig<bool>& F4o4);
boolSig2_F4(int a);


voidF5( const Sig<int>& F5i5, Sig<bool>& F5o5);
boolSig1_F5(int a);


voidF6( const Sig<int>& F6i12, const Sig<int>& F6i13, Sig<int>& F6o9);
intcomputeMomemtum_F6(int mass1, int velo1);


voidF7( const Sig<int>& F7i14, const Sig<int>& F7i15, const Sig<int>& F7i16,
Sig<int>& F7o10);
intcomputeEnergy_F7(int mass2, int velo2, int dist);


voidF9( const Sig<int>& F9i20, Sig<bool>& F9o13);
boolSig5_F9(int a);


voidF8( const Sig<int>& F8i17, Sig<bool>& F8o11);
boolSig4_F8(int a);


voidF2( const Sig<int>& F2i2, Sig<int>& F2o2);
intinputA_F2(int a1);

#endif

/****************************************************************************
/
```

List of Symbols, Abbreviations and Acronyms

*ASIP:*Application Specific Instruction Set Processor

*BDD:*Binary Decision Diagram

*CAD:*Computer Aided Design

*CGS:*Centimetre Gram Second

*CNF:*Conjunctive Normal Form

*CPOG:*Conditional Partial Order Graph

CT: Continuous Time

DE: Discrete-Event

*DSP:*Digital Signal Processing

*FIFO:*First-In First-Out

*FSM:*Finite State Machine

GALS: Globally Asynchronous, Locally Synchronous

*HCDG:*Hierarchical Conditional Dependency Graph

*HDL:*Hardware Description Language

*KPN:*Kahn Process Networks

*MBD:*Model-Based Design

MD-SDF: Multi-Dimensional Synchronous Data-Flow

*MKS:*Metre Kilogram Second

MoC: Model of Computation

*MRICDF:*Multi-Rate Instantaneous Channel-connected Data Flow

*RTL:*Register-Transfer Level

*SAT:*Satisfiablity

*SCADE:*Safety Critical Application Development Environment

*SDF:*Synchronous Data-Flow

*SI:*System International

*SMT:*Satisfiablity Modulo Theory

*TBB:*Threading Building Blocks

*UML:*Unified Modeling Language

*VHDL:*VHSIC Hardware Description Language

*VHSIC:*Very High Speed Integrated Circuit

V&V: Verification & Validation